Real-Time Hand Printed Character Recognition on a DSP Chip

by

Christopher Isaac Chang

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science and Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1995

Auth.....

Department of Electrical Engineering and Computer Science
February 6, 1995

Certified by .....

Professor V. Michael Bove
Thesis Supervisor

Certified by .....

Mansoor A. Chishtie
Texas Instruments Supervisor

Accepted by .....

F.R. Morgenthaler
Chairman, Department Committee on Graduate Students

Real-Time Hand Printed Character Recognition on a DSP Chip

by

Christopher Isaac Chang

Submitted to the Department of Electrical Engineering and Computer Science on February 6, 1995, in partial fulfillment of the requirements for the degrees of Bachelor of Science in Electrical Engineering and Computer Science and Master of Science in Electrical Engineering and Computer Science

Abstract

The study of character recognition algorithms has become increasingly intense due to the growth in areas such as pen-based computers, scanners, and personal digital assistants. One algorithm that has potential commercial applications and that recognizes boxed, hand printed characters is explored and implemented on a TMS320C5x Digital Signal Processing (DSP) chip. By using the DSP chip, the algorithm can fully utilize the chip's special characteristics, and the result is a fast, real-time character recognition system.

The algorithm, when run on a 486 33MHz PC, initially recognizes characters at a rate of 20 ms per character. Once being run on the DSP, the algorithm is capable of recognizing a character in about 14.6 ms. The algorithm was then optimized in the DSP's assembly language to further increase the recognition speed, and a rate of 5.7 ms per character is achieved. Finally, an unboxed recognition algorithm is implemented on the DSP chip for real-time recognition.

Thesis Supervisor: V. Michael Bove
Title: Professor of Media Arts and Sciences

Thesis Supervisor: Mansoor A. Chishtie
Title: Applications Engineer, DSP Applications

Acknowledgments

I, first of all, would like to thank Mansoor Chishtie for guiding me through this thesis and for always being willing to help. I really want to thank Professor V. Michael Bove for offering suggestions to my thesis and especially for being understanding when things were rough. Thanks as well to the DSP Applications group at Texas Instruments, particularly Gene Frantz, Raj Chirayil, and Jay Reimer, for allowing me to work on this project. And, I want to thank the people at Texas Instruments who gave me moral support, including Padma Mallela, Jenny Tsuei, Brian Davison, Mark Nadeski, and in particular, Stephanie Tsuei.

I also need to thank Girolamo Gallo and his group at TI Italy for providing the code, and I appreciate Girolamo's answering all of my questions. Although I never met him, I need to thank Randy Preskitt for writing a thus far error-free EVM communications program.

I want to thank Wilson Chan for helping me with my FrameMaker problems. And I want to thank Kathy Chiu for giving me moral support at MIT. Finally, I want to thank my parents, Mrs. Sue Chang and Dr. C.Y. Chang, and my sister, Joanne. They, more than anyone, have been supportive of my efforts and have inspired me to never give up.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides an overview of the character recognition field. Section 1.1 gives the motivation for studying character recognition. Sections 1.2 and 1.3 explain the two different categories within the field, off-line recognition and on-line recognition. Section 1.4 then gives a brief description of several different algorithms, and this section will serve as a background reference for which to compare the algorithm that is implemented in this thesis. Section 1.5 details the thesis objectives, and finally, Section 1.6 explains the organization of this thesis.

1.1 Motivation

The first known conception of a machine that would recognize characters was in 1929 by Gustav Tauschek, who received a patent in Germany for an optical character recognizer that would recognize pre-written text [1]. Because of the lack of computational resources, he was limited in what he could actually create; his machine used a simple-minded template matching scheme. He placed the input characters under a mechanical character mask and shone light through those masks. A photodetector was placed underneath the input characters, and if the input character matched the character mask, then the light was blocked and therefore did not reach the photodetector. The machine had thus recognized a character [1].

Progress in character recognizers did not improve much as technology remained ineffective and outmoded until the first commercial computer was introduced in 1951 [1]. The advent of the computer age drastically changed the character recognition field and made more sophisticated recognition possible. Intense research began in pursuit of a human-equivalent character recognition machine; the market became flooded with companies interested in creating a recognizer, with all of them driven by the profits that could be made with a machine that could be used to read invoices, typed text applications, zip codes, and preprinted order forms [2]. New strategies were employed to increase the accuracy and flexibility of optical character recognizers that could detect pre-typed text, but these strategies were still limited by the computer technology. The processors were simply not powerful or fast enough to allow for much more flexible and accurate recognition. Recognition was often limited to simple typed text input. Although some research in recognizing hand-printed characters was done, the limitations in technology precluded any serious progress [3]. Thus, by the 1970s, research abated and interest and funding turned elsewhere.

The interest was renewed in the 1980s and has remained intense since then. This rejuvenation was partly due to the better recognition algorithms that were developed, but more importantly, the processors now are more capable of handling the more sophisticated algorithms. With the improvement in technology in computers and in digitizing tablets,

which record pen movements to allow for recognition, real-time or on-line recognition has also been studied intensely by researchers. This on-line recognition allows handwritten characters to be recognized, and this area of research is inspired by both the need to automate office work and also the need to find a less intimidating way to enter text into machines [25].

Commercial products have spawned from this on-line recognition research; most notably, perhaps, the Newton by Apple and the ThinkPad by IBM. The Newton is a personal digital assistant, which is a hand-held device that behaves, in some respects, like a computer but is much more limited in its functionality. It allows the user to record memos, maintain calendars, and keep addresses, and it can do these tasks all electronically and with a pen input. With possible fax and modem capabilities, it allows the user to send and receive faxes, e-mails, and perform such tasks as shopping and banking. Personal digital assistant products such as the Newton, all of which employ character recognition algorithms, have not enjoyed their anticipated success, and the primary reason is their poor character recognition capabilities [4]. Other complaints of personal digital assistants include price, speed of recognition, and size [5].

Another character recognition product is the ThinkPad by IBM. This product is a pen-driven personal computer. Rather than having a mouse or keyboard, it relies on a pen to receive input from the user, and like the Newton, it suffers greatly from poor character recognition and does not remain as a practical choice for consumers [6]. A third type of character recognition product is Microsoft Windows' recently introduced Windows for Pen Computing, which allows applications to be driven by the pen and electronic tablet. Both of these products, though, are criticized for their price, speed, and handwriting recognition accuracy [6]. In all of these products, there are trade-offs that have to be made in terms of price, speed, accuracy, and complexity.

Despite these criticisms of these commercial products, there is much optimism that with improved technology and better algorithms, these products will soon become accepted and desired by the market. In the quest for a machine that can mimic human recognition, the problem that lies ahead is how to resolve these compromises that must be made to introduce an algorithm that is fast and accurate yet whose code is small and can be processed quickly and inexpensively. With the profit gains that can be made in coming with a reasonable solution to this dilemma, many companies are spending considerable time and money to search for a human-equivalent machine recognition system.

1.2 Off-line Recognition

The field of character recognition can be divided into two categories, on-line recognition and off-line recognition [7]. Off-line recognition refers to the recognition of pre-written or pre-typed text. Thus, the information that can be used is only that observed after the writing has been created; this type of information is called optical information. Optical information includes, for example, the height of a character, width of a character, the

presence or absence of descenders (the tail of a character such as in 'g' and 'y') or ascenders (the upward extension of a character such as in 'b' and 'd'), and the relative spatial position of points.

Using only optical information for recognition is called optical character recognition (OCR), and off-line character recognition is a subset of OCR [1]. Although most OCR systems recognize only machine printed characters, there has been a lot of work in off-line recognition of hand-printed text as well. The text, either pre-written or pre-typed, is often inputted to recognition systems through an optical scanner. The input characters are read into the scanner and then digitized into a binary map. Each character is then located and segmented to begin the process of recognizing each character individually. Often times, either the original text given to the scanner, which sometimes suffers from poor resolution due to printer or fax quality, or the process of scanning the text can cause characters to be degraded. Typical degradations are broken characters, random, isolated dots or stray marks, and ligatures. Ligatures consist of two or more characters that are joined together due to the type of font and thus are recognized as a single composite character, as shown in Figure 1.1 [8].

Figure 1.1 The 'f' and 'l' (in Bookman Old Style font) could be misrecognized as an 'A'

In order to eliminate these degradations, the input characters are usually fed into a preprocessor. The preprocessor is designed to fill the holes and breaks in lines, eliminate noise, thin the characters, separate characters, and normalize the size of characters. After this preprocessing, then the actual recognition is performed [9].

In general, there are two primary approaches that have been taken to perform off-line recognition. These two approaches, while at one time distinct, are rapidly converging into one. The first approach is template matching and the second is a structural feature-based approach [10]. Template matching was the first approach taken; there are two basic steps involved in template matching. The first is superimposing a character template onto the input character, and then the second is calculating the difference between the two. This was the approach taken by Tauschek in 1929. The other approach is a structural analysis of characters. This approach arose from the fact that the template matching approach was only useful in the recognition of machine printed characters. With all of the variations of handwritten characters, it was necessary to create this new structural feature-based method. In this method, the goal is to break a character into parts and then determine features of the parts and relationships between those features so that the description can uniquely identify that one character. Thus, this method captures the essential features of characters such as loops, endpoints, curves, joints, concavities, convexities, junctions, and strokes [9]. These two approaches, though, are more often now combined to exploit the

strength of template matching for global analysis of the characters and the strength of structural analysis for detecting local features of characters [11]. Some specific examples of recognition algorithms will be described in Section 1.4 below.

Applications of off-line recognition systems are numerous [12]. The ability to read pre-written or pre-typed text can be used in the post office, for example. Much research has been done and resulting applications have been built that allow scanners to accurately read the zip codes from letters. This time-saving process is one way that OCR can be used. Yet, there are other ways as well, such as with check-readers for banks. The problem with an application such as this is that accuracy must be perfect; otherwise, very costly errors can occur. Other applications range from reading faxes and scanning them into PCs, such as with software applications like WinFax, to simply scanning in data to edit and use on the PC.

Although much research has been done in off-line recognition and OCR, many problems still face the field. First of all, there is the problem with poor printer or fax quality and improper digitization of the input, as mentioned above. The input to the scanner is not always perfect and does not always possess high resolution; lower resolution inputs can result in either holes or breaks in the lines, or random noise can be mistakenly identified as parts of characters and improperly digitized as a '1' in the binary map [13]. Another problem is that which arises from standardizing the characters for input to the recognition system. In order to normalize all of the characters, it is necessary to skeletonize, or thin, the characters. This step is not, however, trivial, and many proposed methods have been attempted to take the binary map of the input text and thin all of the characters [14]. Furthermore, with the problem of ligatures, this process is even more complicated and proper segmentation of characters must be first performed.

Problems also arise with different styles of characters, such as italics, boldface, and underlining. Because these are styles likely to be encountered in text, these problems must be properly circumvented. Italics, for example, cause problems during the segmentation phase because of the difficulty in determining which strokes belong to which character [2], [15]. Finally, a problem with OCR and off-line recognition is the numerous different fonts that need to be recognized. There is no standard font, so all fonts must be able to be handled and recognized properly by the OCR system. Although many commercial OCR packages claim to be omnifont, meaning they can recognize all characters by extracting features to ensure that characters are recognized correctly even if they only remotely resemble the actual character, the accuracy of those packages is not high [16]. Thus, training is still often required, and this training can be a very time consuming and tedious process. Furthermore, the training requires a lot of memory to create a database for each font required to be recognized [8]. These problems still leave much area open for active research in OCR and off-line recognition.

1.3 On-line Recognition

The other type of character recognition is on-line recognition [7]. This type refers to the recognition of characters as they are written; i.e., the recognition is done in real-time. Thus, not only can the optical information be used, but also the dynamic information can be used to recognize characters. The dynamic information is the temporal information that is obtained from being able to observe characters as they are written. This information includes the number of strokes taken to write a character, the order of strokes, the direction of the strokes, and the relative temporal position of points and strokes.

In on-line recognition, there are many types of handwriting that can be recognized. Examples of the different types are shown in Figure 1.2. For all of the handwriting types except for boxed recognition, a segmentation routine is required to separate the letters from one another and associate each stroke with a character. Segmentation is performed by the user in boxed recognition by writing each character in a separate box. Segmentation for the other types of handwriting recognition is not a trivial process, and it becomes exceedingly difficult for the cursive script writing compared with the discrete types of writing. This difficulty arises from the fact that in the discrete writing cases, each character is still composed of one or more strokes. Thus, segmentation can occur between strokes. For the cursive script case, however, segmentation must be performed within a stroke, for a single stroke can be used to write several characters. Segmentation routines are still being proposed for cursive script, but much work needs to be done in this area to improve the recognition results.

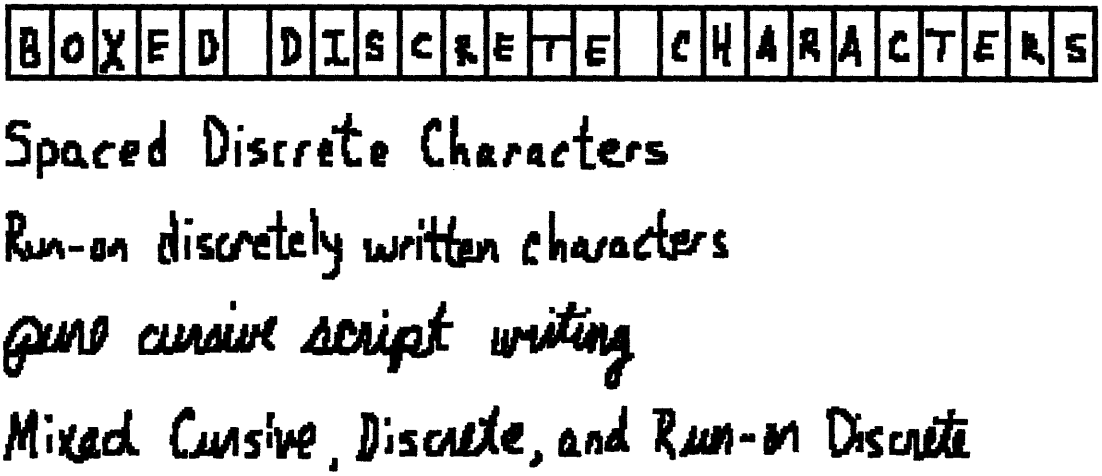


Figure 1.2 Types of handwriting character recognition [7]

With on-line recognition, it is necessary to have a device to capture the strokes as they are written. Digitizing tablets are used for this real-time capture of the coordinates of each character. There are currently two main types of digitizing tablets, electromagnetic tablets and pressure sensitive tablets [7]. The electromagnetic tablets have grids of conductors underneath the writing surface or screen. The writing pen, or stylus, usually contains a

loop of wire that enables it to tell the tablet its location. In some electromagnetic tablets, the pen is active and sends the pen's position back to the grid, while in other tablets, the grid is active and acts as an antenna to receive a magnetic signal from the pen. The other main type of tablet is a pressure-sensitive or resistive one. These resistive tablets have layers of resistive and conductive material underneath the writing screen. There is an electric potential across one of the resistive layers in one direction, and there is thus a voltage gradient that is directly proportional to position. Then, when the stylus comes into contact with the screen, the voltage can be read and the position can thus be determined. These two types of tablets are the most common; the resistive tablets tend to be less expensive and do not require a special stylus, but the electromagnetic tablets tend to be more accurate and more flexible. In order for any tablet to be useful in real-time recognition, it is critical that they have a resolution of at least 200 points/inch and a sampling rate of at least 100 points/second [7]. These guidelines enable enough data to be collected and shown to uniquely capture and identify characters written on the tablet.

After the real-time data is captured by the tablet, the next step is to send those coordinates constituting each stroke to a preprocessing unit to standardize the input, similar to the stage in the off-line recognition systems. In this stage, one step is to smooth the character to eliminate erratic bumps caused by uneven hand motion; another step is to thin the character to eliminate redundant points and to interpolate when there are not enough points; the goal of this step is to produce a character that has a relatively equal distance between each point. An additional step is to connect strokes that were disconnected due to accidental pen lifts. Other purposes of the preprocessing of real-time recognition characters are to eliminate random points caused by accidental pen-down, to get rid of hooks at the end of strokes caused by accidental pen-up or pen-down markings on the screen, to correct and straighten vertically and horizontally slanted characters, and to normalize the size of characters [17], [18]. Thus, the preprocessing stage, although similar to the off-line preprocessing stage, performs tasks that are designed to fix the abnormalities observed during and caused by the real-time writing of characters on a tablet. After the preprocessing stage, the final step is to recognize the characters. The recognition algorithms used can be the same as those used in off-line recognition systems, with the additional dynamic information being used, and Section 1.4 below describes several approaches to character recognition.

Because the recognition is done in real-time, there is a constraint on the speed of the algorithm. Whereas in off-line algorithms, recognition can occur at any speed, in on-line algorithms, the recognition must be able to keep up with the writer's writing pace. This requires that the code cannot be too complex and that the processor must be able to process the input data quickly.

Despite this restriction, the advantages of on-line recognition are numerous [7], [19]. Using solely optical information can be limiting in the approaches to recognition and can severely diminish the accuracy rate. With the additional dynamic information, the recognizer can now use the direction of the pen, the order of strokes, and the path of the pen to aid in recognition. Furthermore, the costly and imperfect step performed in off-line

recognition of skeletonizing the character is no longer necessary, as the output from digitizing tablets is the series of coordinates making up each stroke. Thus, with the temporal information along with the optical information, the recognition accuracy can be significantly increased.

An additional benefit of on-line recognition is that there can be interaction between the user and the recognizer [7]. The writer can write characters on the digitizing tablet and verify recognition on the screen. Thus, if the recognizer misrecognizes a character, the user receives instant feedback and can correct the recognizer immediately and thus teach it. Another possibility is that the user can adjust the way he or she writes a character and thus adapt to the recognition ability of the system. On-line recognition also allows the user to train the recognizer to adapt to and recognize his or her writing style. Thus, once the recognizer is trained to recognize a user's style, recognition rates generally increase dramatically. This interactiveness provides a big improvement over off-line implementations.

Applications of on-line recognition systems include the personal digital assistants and pen computers mentioned above. There are numerous other applications of on-line recognition [20]. One such application is used for filling out forms. Rather than filling out the forms on paper and then entering the data into a computer, the data can be directly written on and processed in the computer. Another application might be for editing and processing text. Because of the high amount of interactivity in editing papers, on-line recognition is ideal for this purpose. Finally, applications for on-line recognition systems exist already in mobile services where a computer is necessary and are used by people such as UPS drivers or police officers [7]. In these cases, the user can take notes and fill out forms on the screen directly with a stylus, without having to use a cumbersome keyboard or mouse. Thus, applications for real-time recognition systems already exist, with the need for improved recognition algorithms to further develop more significant and diverse applications.

1.4 Character Recognition Algorithms

There are many different approaches to recognizing hand printed characters. This section will attempt to give a brief overview to the character recognition field by highlighting several approaches and briefly describing each of them. This is by no means a complete overview of all of the algorithms, but instead, it is a sampling of current strategies and approaches to the recognition of hand printed characters. The final subsection of this section does not describe a conventional character recognition approach to recognizing text. This subsection instead describes the use of the Hidden Markov Model for word recognition rather than just character recognition; it is included in this section to illustrate an alternative approach to recognizing text compared to more conventional character recognition methodologies.

1.4.1 Template Matching Approach: Similarity Measures

Template matching is a global comparison of an input character with a reference character from the database, in terms of points or groups of points. One type of template matching is described in [21], in which input character digits are compared with reference characters on a point by point basis. Input characters are compared with templates by using the Jaccard and Yule similarity measures separately. These measures are defined in the following way. Let T and D be binary arrays representing a template and a digit, respectively. Then, let n_{ij} = the number of times that T(x,y) = i and D(x,y)=j for i,j=0,1. Thus, n_{10} = the number of times that T(x,y) = 1 and D(x,y)=0 for all corresponding (x,y). Then, the Jaccard and Yule similarity measures are defined as:

$$\text{Jaccard: } \frac{n_{11}}{n_{11}+n_{10}+n_{01}}$$

$$\text{Yule: } \frac{(n_{11}n_{00}-n_{10}n_{01})}{(n_{11}n_{00}+n_{10}n_{01})}$$

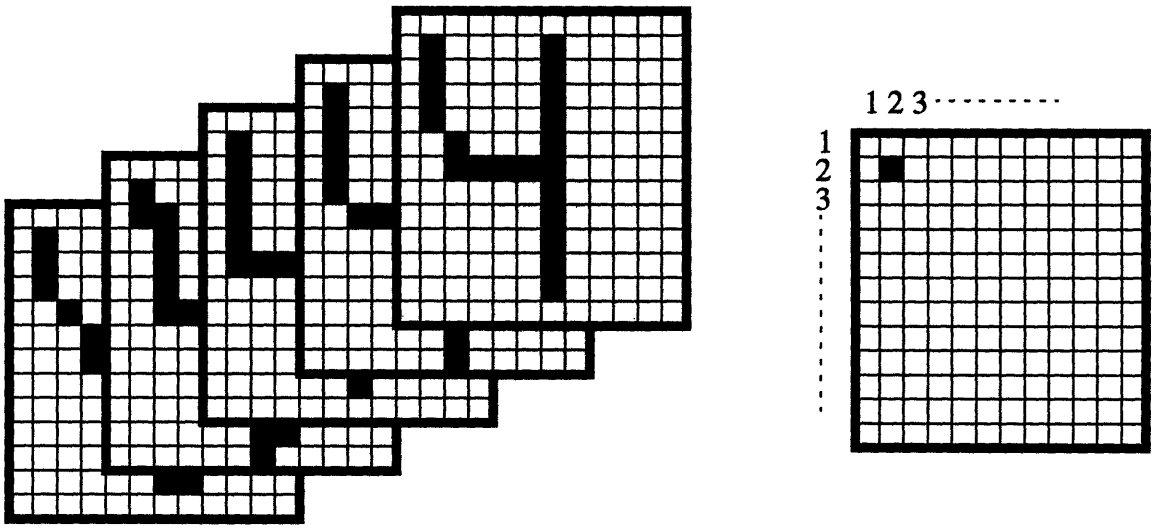
The first step in this method is thus to create the templates for which to compare the input character. The templates are created in the following manner. A sample of digits written in various styles is taken and normalized in terms of size and moments. After this normalization, templates are constructed in two steps. First, a sequential cluster of potential templates is created, and second, a single template is extracted from each sequential cluster. The sequential clustering step involves comparing a digit from the sample with each digit in each cluster. Both the Jaccard and Yule similarity measures are calculated, and if the highest similarity measure, between the input character and one of the characters in the clusters, is above a threshold, X, then the input character is placed in that cluster. Otherwise, a new cluster is created. At the end of this process, there are a varying number of clusters for each digit, with each cluster containing a varying number of binary arrays. From this sequential cluster, a single template is extracted for each digit by constructing a new binary array for each digit based on the clusters. The rule used is that if P% of the digits in the cluster have a value of '1', indicating a black pixel, at the location (x,y), then a '1' is placed at (x,y) in the binary array. This step continues for all (x,y) and the end result is a binary array that represents the digit; i.e., a template is created. Furthermore, outliers can be eliminated by not using those clusters containing fewer than M digits in them. With the parameters being able to be user-defined, this system allows very high flexibility in the creation of clusters and templates. This creation of templates is shown in Figure 1.3.

Once the templates are created, then the process of recognizing new digits simply involves performing the Yule and Jaccard similarity measures between the input character and the templates and then using some sort of decision rule to determine the correct character. The decision rules described in [21] are one way to select the correct character with very high accuracy.

This template matching approach does suffer, though, from the problem of not being

rotation invariant. Thus, if a digit is slanted or slightly rotated, there is the possibility that the binary arrays of the input character and the templates will not yield very high similarity measures, and thus the input character can either be misrecognized or not recognized at all. This problem is a common one among many template matching schemes.

Rule: If 45% or more of the binary maps in a cluster have a '1' at (x,y) then place a '1' at (x,y) in the template



Cluster of '4's

Template for '4'

Figure 1.3 Because more than 45% of the binary maps in the cluster have a '1' at (2,2), then a '1' is placed at (2,2) in the template that will be used to represent the number '4'

1.4.2 Global Approach: Fourier Descriptors

Another approach to the recognition of characters is the use of a transformation to describe and represent a character. In some respects, the use of a transformation is a type of template matching, as the global features of the character are used for recognition, while none of the local features or structures are examined and used. This algorithm is included separately in order to illustrate the use of invariant transformations. One typical transformation used is the Fourier transformation, and from this, Fourier descriptors are used to recognize characters. The Fourier descriptors are selected because they have rotation, translation, size, and shift invariance. These properties are desirable because of the problems described in template matching where a rotation in the character can cause a misrecognition. These transformations eliminate those differences and allow proper recognition of characters. Furthermore, the use of Fourier descriptors to describe a character provides the added benefit of significant data reduction; rather than storing the

entire coordinates of the character, the descriptors can be calculated and stored, thus saving memory.

One method of using the Fourier descriptors to recognize digits, as described in [22], [23], and [24], is to take the Fourier transform using the coordinates of the boundary of the character. In order to correctly obtain the boundary, it must be ensured that the boundary is closed and continuous, and the boundaries of consecutive characters should be properly segmented. Any type of boundary algorithm can be applied to the input characters. A further guideline is placed on the boundary search in that consecutive coordinates of a boundary are not diagonal to one another; i.e., diagonal neighbors are not allowed. This constraint aids in the calculation of the perimeter.

Thus, a series of L coordinates describes the boundary of a character such that x(1)=x(L) and y(1)=y(L). Then the Fourier coefficients can be calculated from these coordinates (x,y) from the following two equations:

$$a(n) = \frac{1}{L-1} \sum_{m=1}^{L-1} x(m)\, e^{-jn\omega_0 m}$$

$$b(n) = \frac{1}{L-1} \sum_{m=1}^{L-1} y(m)\, e^{-jn\omega_0 m}$$

where $\omega_o = 2\pi/L$. These coefficients, however, are not rotation or shift invariant. Thus, a set of invariant descriptors are calculated as

$$r(n) = [|a(n)|^2 + |b(n)|^2]^{\frac{1}{2}}$$

so that it is now rotation and shift invariant. Finally, to ensure size invariance, a new set of descriptors, s(n), are calculated by

$$s(n) = r(n)/r(1)$$

These Fourier descriptors, s(n), can now be used to completely describe the character; furthermore, in [24] it is shown that for almost all characters, typically only five or six of these Fourier descriptors have significant values. Thus, a serious data compression can be achieved.

Fourier descriptors can be calculated for all characters in a database. Then, to recognize an input character, the Fourier descriptors are calculated for that input character. As mentioned in [23], two closed curves which differ only in position, orientation, or size with analogous starting parts have identical Fourier descriptors. Thus, the starting points

for all characters must be normalized before the Fourier descriptors are found. Once the starting points are normalized and the Fourier descriptors are calculated, it is shown in [24] that the best measure of similarity between the input character's Fourier descriptors and those from the database is to use the squared Euclidean distance:

$$D_{mx} = \|F_m - F_x\|^2$$

where F_m is the reference feature vector containing the Fourier descriptors of character m from the database, and F_x is the feature vector of the input character x.

The correct identifier is the character from the database with the smallest squared Euclidean distance from the input character. Because of the rotational invariance, though, problems arise from the recognition of characters such as '6' and '9' or '2' and '5'. Thus, a feedback routine can be applied to examine those characters more carefully and determine the correct identifier. Furthermore, subclasses of characters have to be created because of the different ways of writing the same character. With these subclasses and feedback, the recognition accuracy achieved through the use of Fourier descriptors proves to be very high and very effective for the recognition of numbers.

1.4.3 Feature Based Approaches

Another approach to recognizing characters, and by far the most popular, is the use of features to describe characters [9]. These features, whether geometrical, topological, or by distribution of points, enable a very accurate description of characters with relatively high data compression. Some examples of geometrical and topological features are intersections of line segments, presence or absence of loops, presence or absence of dots, presence or absence of descenders or ascenders, stroke relations, angular properties, and directions of strokes and curves.

The use of distribution of points as features is also common. This can include using moments of black pixels about the center of the character, using horizontal or vertical zero crossings on the axes about the center of the character, and using a method like zoning or characteristic loci. Zoning involves splitting the character image map into non-overlapping regions and then calculating the density of points in each region. If the regions are chosen properly, then these densities can make up a feature vector that uniquely identifies each character to be recognized. Characteristic loci involves creating, for each white pixel, vertical and horizontal vectors. Then, the number of times the line segments making up the character are intersected by these vectors is used as the feature vector for the character [9].

Thus, there are numerous types of features that can be used or multiply used to characterize a character. The goal of this process is to uniquely identify all characters while minimizing the amount of data necessary. The strength of this approach is not only its relative simplicity and lack of complexity necessary, but also its tolerance to distortion, style variations, translation, and rotation. Furthermore, although tolerant of variations, it

is still not too tolerant so as to not uniquely define characters. These methods do, though, require proper and accurate preprocessing in order to normalize characters before extracting features.

1.4.4 Other Approaches to Character Recognition

There are many other approaches to character recognition besides the ones mentioned above. The method described in [7] uses a technique for real-time recognition called analysis-by-synthesis. In this method, strokes are the basic elements, and rules for connecting the strokes are used to build symbols. The symbols that are created from these rules are considered the character database for recognition; then, mathematical models can be created for each stroke to model the direction of the pen as a function of time as each stroke is written. Thus, to recognize a character, it simply has to be divided into strokes, and the strokes can then be classified according to the model parameters and the character will be recognized.

Another approach, described in [25], is called a morphological character recognition method. In this method, a character, or shape, is reduced to an axial representation using the morphological skeleton transform (MST). The MST compacts the shape into a basic skeleton while still retaining all of the necessary information of the object's shape and size. The basic premise of this type of recognition is that the skeleton of a character is all that is necessary for unique identification of characters. The MST is translation invariant, but in order to maintain size and rotation invariant, the character must first be normalized. Once the character has been transformed using the MST, the recognition then occurs with a skeleton matching algorithm (SMA) that measures the distance between the input character and the character from the database.

Similar to the Fourier transform, another transformation that is used is the Walsh functions [26]. The Walsh functions are orthonormal functions that can be used to represent the character data. They have the benefit of not requiring any preprocessing to be performed on the input character other than digitization of the input character image. Thus, computation time normally spent during the preprocessing stage of digitization, thinning, and noise removal is greatly reduced. The size-normalized digitized character is scanned [redacted] specific direction. This intensity distribution function can then be used along with a user-defined number of Walsh functions, depending on the accuracy desired, to calculate coefficients that can uniquely define characters.

Finally, neural networks have been used in character recognition [26], [27], [28]. Using neural networks is not a method of recognizing characters, but rather, they are one way of implementing a character recognition system. Neural networks operate by attempting to mimic the brain's neural system behavior. They can be used to generalize and correctly recognize input characters that were not necessarily learned during training. The neural networks can be trained to learn specific features of characters through techniques such as backpropagation, and the networks can then be used to recognize a wide range of characters, as described in [27]. The neural networks have the advantage of greater

flexibility, and with the use of parallel processors, faster recognition. However, neural networks still suffer from the problem of high computational power required and convergence problems in the backpropagation technique [27].

1.4.5 Word Recognition: Hidden Markov Model

The Hidden Markov Model has been used for hand printed word recognition. In [29] and [30], a first and second order Hidden Markov Model based approach is used to recognize text. Rather than recognizing one character at a time, this method approaches the problem of recognizing text by examining an entire word and trying to determine the best combination of letters to select the correct word. The method works by taking an entire word as input. The word is then segmented into individual characters, and features are extracted from each character. Then, using a codebook that is created as described below, possible sequences of characters are produced. Finally, given these possible sequences, the Viterbi algorithm is used to select the most probable sequence as the recognized word. This approach is mentioned to illustrate a typical feature extraction process and also to describe a different approach to the recognition of characters for the purpose of recognizing text.

Thus, after segmentation, the first step is to extract features from the characters. In the algorithm described in [29], the features to be extracted were chosen in order to minimize computation, independently characterize each character, and be independent of rotation, translation, and size. Thus, one set of features used is based on the central moments of a character. The coordinates of the 2-D binary image are used to calculate the central moments, and then three features based on these moments that are size, rotation, and translation independent are used. The central moments are given by the following equations, for the coordinate of a character expressed as (u,v):

$$\mu_{pq} = \frac{1}{N}\sum_{i=1}^{N}\left(u_i - \bar{u}\right)^p\left(v_i - \bar{v}\right)^q$$

where

$$\bar{u} = \frac{1}{N}\sum u_i; \qquad \bar{v} = \frac{1}{N}\sum v_i;$$

and N is the total number of black pixels in the binary image of the digit. The three features extracted from the central moments are:

$$M_2' = \frac{M_2}{r^4} \qquad M_3' = \frac{M_3}{r^6} \qquad M_4' = \frac{M_4}{r^6}$$

where

$$r = (\mu_{20} + \mu_{02})^{\frac{1}{2}}$$

$$M_2 = (\mu_{20} - \mu_{02})^2 + 4\mu_{11}^2$$

$$M_3 = (\mu_{30} - 3\mu_{12})^2 + (3\mu_{21} - \mu_{03})^2$$

$$M_4 = (\mu_{30} + \mu_{12})^2 + (\mu_{21} + \mu_{03})^2$$

Other features used are the number of loops in a character, the number of zero crossings through the horizontal line drawn through the center of the character and the number through the vertical line drawn through the center, and the presence or absence of dots such as in 'i' and 'j'. After these features are calculated, they are compared with the codebook.

The codebook is created by a vector quantization algorithm. The features described above are stored in a feature vector, and this feature vector is created for each character in a sample of letters. A wide sample is necessary to accurately reflect various handwriting styles. Once the vectors for all the letters are created, the next step is to find the set of M vectors, which will constitute the codebook, that will most accurately represent the entire sample of characters. The goal is to find the set of feature vectors such that the average distortion of the quantization is minimized. The average distortion is calculated by the difference between each feature vector in the sample and the closest vector in the codebook. The size of the codebook is determined experimentally and the authors in [29] have shown that a point is reached where increasing M does not decrease the distortion significantly.

Thus, the feature vectors of the input characters are compared to those in the codebook. One or more choices are selected as possible identifiers for each input character. The next step is to calculate the necessary parameters for the Hidden Markov Model. In this model, each letter in the alphabet is a state, so there are thus twenty-six states. The necessary parameters for the first order model are defined below, where q_j and q_i are states and v_k is a symbol from the codebook.

State transition probability: $p(q_j \text{ at } n+1 \mid q_i \text{ at } n)$

Initial probability: $p(q_j \text{ at } n=1)$

Symbol probability: $p(v_k \text{ at } n \mid q_j \text{ at } n)$

These parameters are calculated in the following manner for the first order Hidden Markov Model. The state transition probability is calculated from an extensive study that examined the English dictionary and determined the number of times that one letter followed another for all combinations of letters. The initial probability is calculated by determining the percentage of words in the dictionary beginning with each character. Finally, the symbol probability is calculated by determining the probability that a character written in the sample is identified as the symbol being examined.

The second order Hidden Markov Model uses the same initial probabilities and symbol probabilities, but the state transition probabilities are defined differently. This probability is the probability that the next character is a certain character, given the previous two characters.

Once the probabilities are calculated, the final step is to use the Viterbi algorithm to accurately determine the most likely sequence of characters to recognize a word, given the number of characters and the possible identifiers for each character in the word. The algorithm determines the most likely sequence of characters using the first and second order probabilities described above, and it outputs the recognized word. In [29], it is shown that this form of recognition, though very computationally expensive and memory expensive, yields very high accuracy rates when recognizing words written in various handwriting styles.

1.5 Thesis Objectives

The objective of this thesis is to create an on-line recognition system by implementing a hand printed character recognition algorithm in real-time on a digital signal processing (DSP) chip. The DSP chip is used because of its ability to process mathematically and data intensive code very quickly; this speed is due to its special architecture and instruction set. Though much more limited than a typical microprocessor or microcontroller in terms of what it can and cannot do, the DSP chip can perform a limited set of specific instructions at a significantly faster rate. As a result, these chips are ideal for many real-time applications. Furthermore, with the DSP chip already able to efficiently perform modem, fax, and other telecommunication processes, the chip is thus a very diversified processor and provides the basis for an attractive, integrated platform that can be used for various user-interactive services.

The original algorithm used in this thesis is designed to recognize hand printed characters in a boxed environment. The constrained environment forces the characters to be segmented and limits the characters in size and position. The recognition system is user-trainable, and it provides the capability of inserting and deleting characters from the database. The code is compact, and yet the recognition accuracy is very high, especially after the user training. The algorithm uses a feature-based approach to recognize characters, as well as using the dynamic information generated from the on-line implementation.

The first step in this thesis is to thus implement this algorithm on a DSP chip for fast, real-time recognition. This step requires a good understanding of the algorithm, the DSP chip, and the platform on which the DSP chip is located. Upon implementing the algorithm on the chip, the next step is to optimize the recognition speed. The purpose of this step is to utilize the DSP's special architecture and instructions to make the recognition of characters as fast as possible. This thus ensures that any recognition using this algorithm that is performed off-line is done as fast as possible. This step is also critical in ensuring

that real-time operation does not constrain the speed at which the writer writes and provides no limitations on recognition. In order to successfully optimize this algorithm on the DSP, it is necessary to fully understand the DSP chip, the DSP assembly instruction set, and the DSP's C compiler. Finally, the last stage of this thesis is to enhance the algorithm to allow for more freedom of input. Thus, an unboxed recognition system is implemented on the DSP. This unconstrained environment allows characters to be written outside the boxed environment and allows more freedom in the spacing between the characters. The algorithm still, however, requires that the characters be spaced and thus segmented by the user; it cannot recognize cursive script.

In this thesis, a fully functional implementation of a highly accurate hand printed character recognition algorithm is thus created on a DSP chip. The real-time recognition is optimized for speed, and the implementation allows for very fast and accurate recognition of characters while still being efficient in terms of code size and complexity. These characteristics allow for very high flexibility in applications, and the algorithm is thus an ideal and viable solution for many commercial products.

1.6 Organization

Chapter 1 of this thesis is a basic introduction to the thesis and provides an overview of character recognition technology and algorithms as it stands today. Chapter 2 then describes the digital signal processing chip and the platform that is used, and it discusses the uniqueness of the chip and the reasons it is well-suited for real-time recognition. Chapter 3 details the boxed-environment hand-printed recognition algorithm used. Chapter 4 explains the implementation of the algorithm on the DSP chip, and Chapter 5 describes the optimizations made to the algorithm to speed up recognition. Chapter 6 then describes the enhancements that were made to allow for unboxed recognition, and Chapter 7 provides final comments and suggests areas of further research.

Chapter 2

The DSP and EVM

The purpose of this chapter is to provide an overview of the digital signal processing (DSP) chip and the evaluation module (EVM) used in this thesis. Many of the features of the DSP chip and EVM that will be referred to in later chapters will be discussed here, and this chapter will provide a basic understanding of the EVM, the chip, and their usage.

2.1 The TMS320C5x family

The DSP chip used in this thesis is Texas Instruments' TMS320C50 chip. This chip is one of many chips in the TMS320C5x family, Texas Instruments' latest generation of fixed-point processors. The fixed-point processors are 16-bit chips; there are also floating point processors, which are 32-bit chips. The 16-bit chips provide a fixed level of arithmetic precision, and thus the arithmetic is more like integer arithmetic than floating point arithmetic. The range of numbers that a fixed-point processor can handle is less than that of a floating point processor, so underflows and overflows must be taken into consideration. This smaller range of numbers provides the advantage, though, of relative simplicity and greater ease of programming than with floating point processors.

The TMS320C5x family can be used in a host of applications, as the DSP chips can process mathematically and data intensive code relatively quickly. This quickness allows for many real-time implementations; the range of areas in which the TMS320C5x family can be used includes image processing, speech processing, control systems, telecommunications, consumer electronics, and automotive products. Example applications include echo cancellation for cellular telephones, image compression such as JPEG, vibration analysis for automobiles, speech recognition, and disk drive controllers. This diverse spectrum of applications reflects the DSP's own versatility. The DSP has grown from being able to only perform basic digital signal processing techniques such as filtering and convolutions to being capable of handling a much wider range of tasks. There are still, however, many tasks the DSP cannot perform compared to, for instance, PC microprocessors. Although the DSP is more limited in what it can and cannot do than the standard PC microprocessor, its specialty allows the DSPs to perform specific functions with a much higher speed than the more diverse but less specialized microprocessors can achieve. The TMS320C5x family provides a very flexible instruction set, inherent operational flexibility, and high-speed performance; yet, the chips are very cost-effective and economical in size, making them ideal for commercial applications. The DSP processors are inexpensive alternatives to custom-fabricated VLSI and multichip bit-slice processors as well as standard PC microprocessors.

The TMS320C50 has features that make it compatible with the C language. There are twenty-eight core CPU-related registers that are memory mapped. Included are eight auxiliary registers that allow the C compiler to address them directly as well as provide for

a more efficient addressing mode. The C compiler is a full implementation of the ANSI standard and it outputs optimized assembly language source code. The C compiler supports in-line assembly, and variables defined in C can be addressed in the assembly code and vice-versa. The results of this software package allow the user to take advantage of the higher level language while optimizing for efficiency in assembly language.

For the reasons mentioned above, the TMS320C50 digital signal processing chip was selected to be used in this thesis as the primary engine to perform the real-time hand printed character recognition. The special features and functions that the TMS320C50 chip possesses and can employ will be discussed in the next sections [31].

2.2 The TMS320C50 Architecture

The TMS320C50, like all chips in the TMS320 family, has a modified Harvard architecture. In the strict Harvard architecture, the program and data memories occupy separate spaces, which permits a full overlap of instruction fetch and execution. The modified architecture provides the further capability of allowing data transfers between the program space and data space, which increases the flexibility and increases the processing capability of the device. Along with the ability to transfer data memory to program memory, there is also the ability to initialize data memory from program memory. The chip can thus time-division multiplex its memory between tasks as well as initialize its data memory with constants and coefficients stored in the system's program ROM. This minimizes the system cost by eliminating the need for a data ROM and maximizing the data memory utilization by allowing dynamic redefinition of the data memory's function. The most important reason, however, for basing the chip on the Harvard architecture is speed. The full overlap of instruction fetch and execution effectively doubles the algorithm's throughput.

The architecture provides a four-deep pipeline. Figure 2.1 shows the different stages of the pipeline for single-word, single-cycle instructions executing with no wait-states.

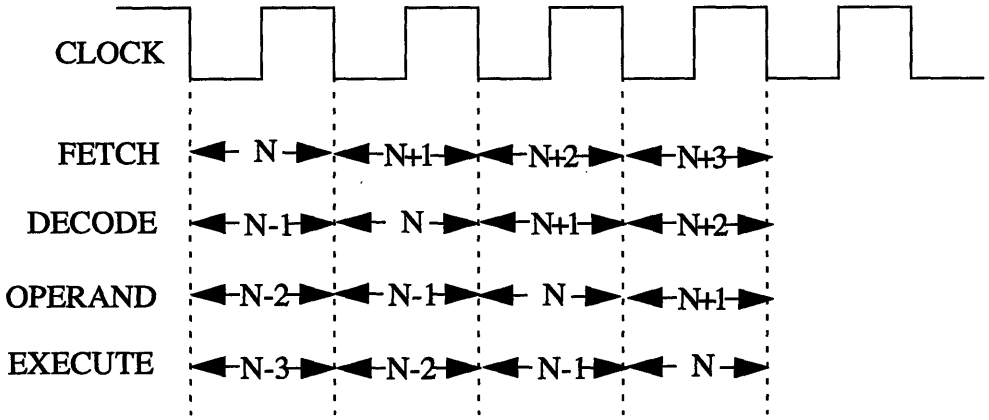


Figure 2.1 Pipeline Operation

As the figure shows, the four stages are the fetch stage, the decode stage, the operand stage, and the execute stage. Each of the four stages are independent, which allows overall instruction executions to overlap. Thus, during any given cycle, four instructions can be active, each at a different stage of completion. The pipeline is essentially invisible to the user, except in such cases where the pipeline must be protected and a latency be added because of data conflicts or pipeline violations. During the fetch stage, the instruction code is loaded. Then, in the decode stage, the instruction code is processed and determined; in the operand stage, the operands are loaded and finally during the execute stage the instruction is performed. The pipeline is useful for delayed branches, calls, and return instructions.

The architecture revolves around two major buses: the program bus and the data bus. The program bus carries the instruction code and immediate operands from the program space, while the data bus interconnects various elements, such as the registers, to the data space. Both the data bus and the program bus are 16-bits wide. This architecture allows a 25 nanosecond fixed-point instruction execution time; a simplified block diagram of the TMS320C50's architecture is shown in Figure 2.2.

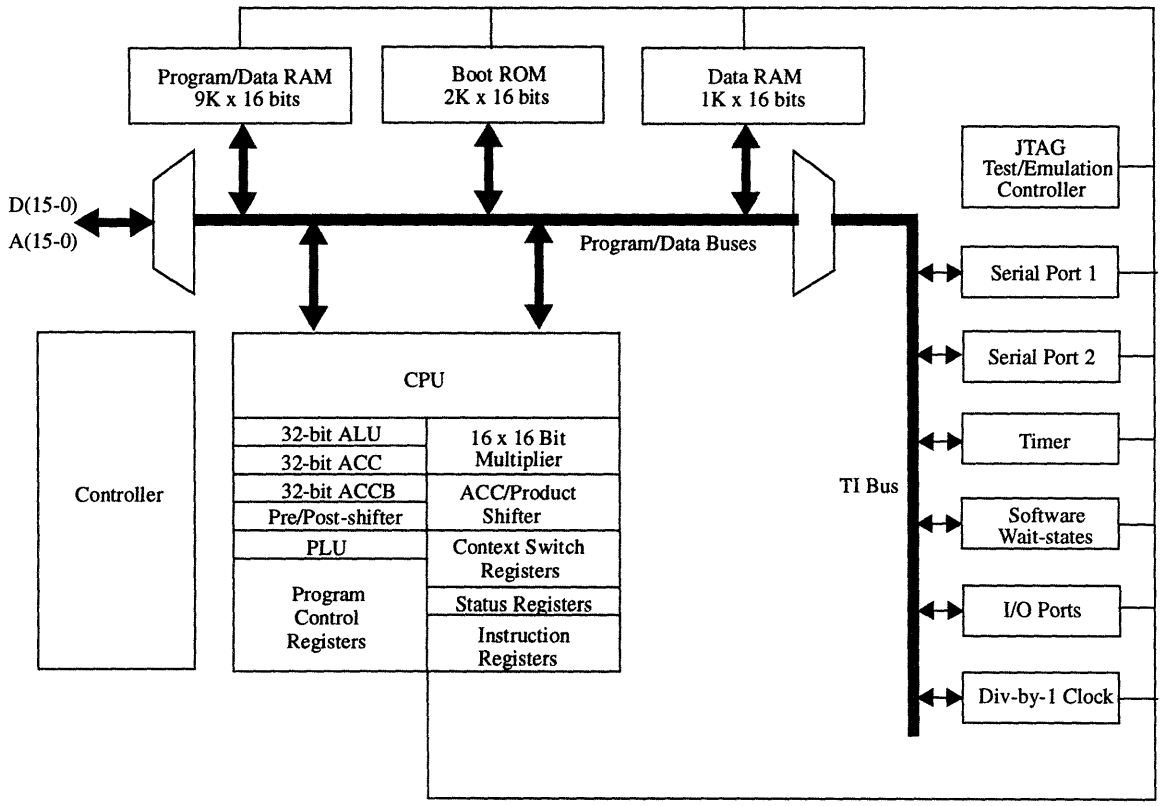


Figure 2.2 TMS320C50 Architecture Block Diagram

2.2.1 Central Processing Unit

The primary block within the chip's architecture is the central processing unit (CPU). The

CPU incorporates the central arithmetic logic unit (CALU), the parallel logic unit (PLU), and twenty-eight program control registers. The next sections will detail the different units within the CPU.

2.2.1.1 Central Arithmetic Logic Unit

The central arithmetic logic unit (CALU) is shown in Figure 2.3.

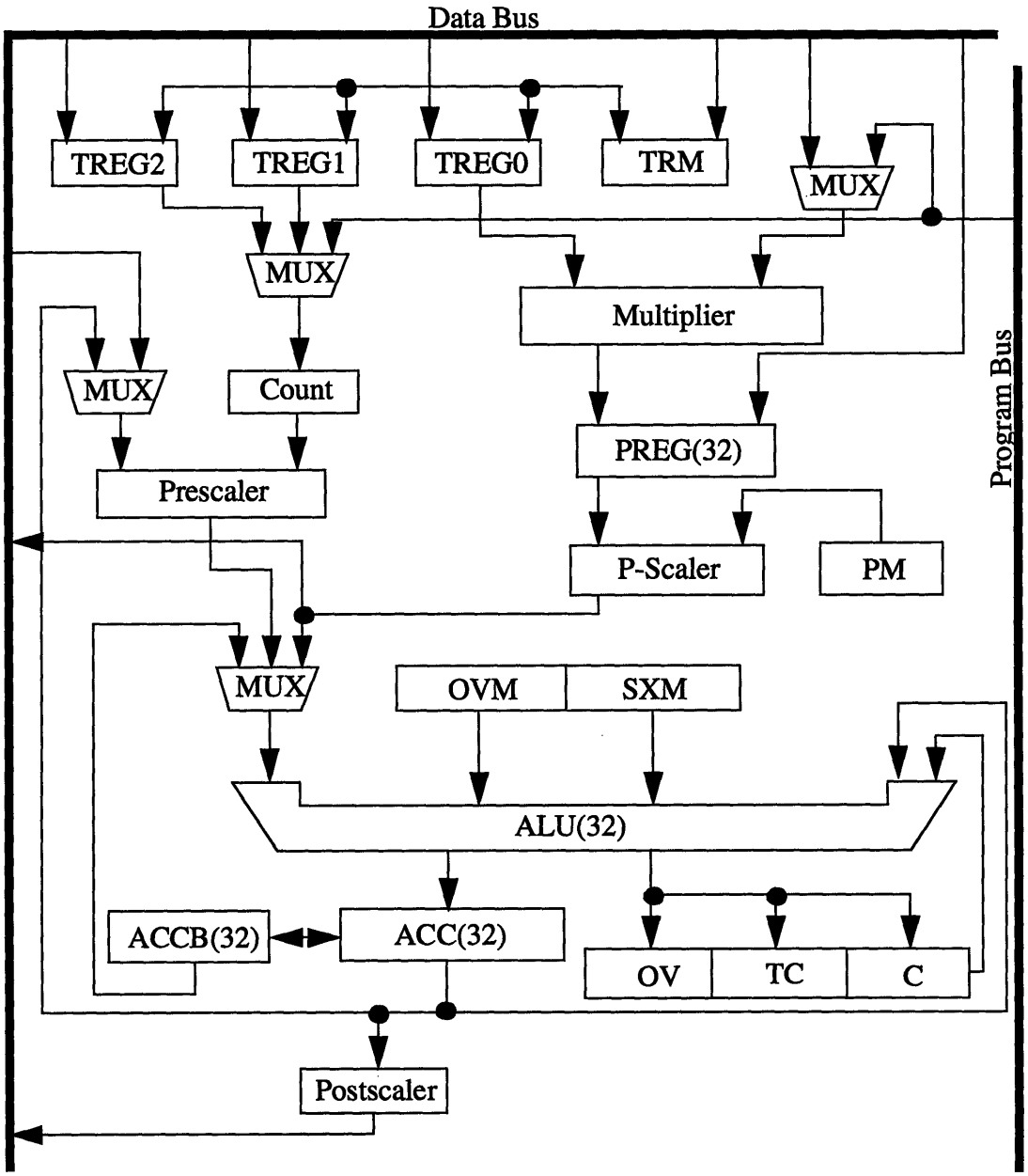


Figure 2.3 The Central Arithmetic Logic Unit Block Diagram

The CALU includes a 32-bit arithmetic logic unit (ALU), a 32-bit accumulator (ACC), a 32-bit accumulator buffer (ACCB), a 16 x 16-bit hardware multiplier, a 16-bit scaling

shifter, and additional shifters at the outputs of both the multiplier and accumulator.

The main component within the CALU is the arithmetic logic unit (ALU). The ALU is a general purpose arithmetic unit that performs single-cycle, 16- or 32-bit logical or arithmetic operations. During the operation, one input must come from the accumulator. The other input can come from the product register (PREG) of the multiplier, the accumulator buffer (ACCB), or the scaling shifter that is loaded from data memory or the accumulator. The ALU and the accumulator implement a wide range of operations, most being performed in a single 25 ns cycle.

In addition to the usual arithmetic instructions such as addition and subtraction, the ALU can perform Boolean operations that facilitate the bit manipulations that are required of high-speed controllers. After the ALU performs the operation, the result is stored in the 32-bit accumulator. The accumulator can be split up into two 16-bit segments for storage in data memory. Recall that the data bus is 16 bits long, and each register and each data location is one 16-bit word long. The result in the accumulator can be shifted when transferring it to the data bus for storage without affecting the contents of the accumulator. This shifting capability aids in bit manipulations and scaling as well as in aligning numbers for extended precision arithmetic, and there are various status bits associated with this shifting. The result can also be stored in the 32-bit accumulator buffer, which provides a temporary storage place for a fast save of the accumulator. The accumulator buffer, as mentioned, can also be an input to the ALU, for operations such as value comparisons between the number in the buffer and the number in the accumulator.

In summary, for all ALU instructions, the following steps will occur:

1. Data is fetched from the data memory on the data bus
2. Data is passed through the scaling shifter and the ALU where the arithmetic is performed
3. The result is moved to the accumulator (ACC)

The CALU also includes a pre-scaling shifter that has a 16-bit input connected to the data bus and a 32-bit output connected to the ALU. The scaling shifter produces a left shift of 0 to 16 bits on the input data. The chip contains several other shifters as well that allow it to perform numerical scaling, bit extraction, extended-precision arithmetic, and overflow prevention. These shifters are connected to the output of the product register and the accumulator. There is a post-scaling barrel shifter from the accumulator and the accumulator buffer to the data bus, a product shifter for scaling the multiplier results, and an accumulator barrel shifter capable of a 0- to 16-bit right shift.

The last primary element in the CALU is the 16 x 16-bit hardware multiplier that is capable of computing a signed or unsigned 32-bit product in a single 25 ns cycle. Almost all multiplications perform a signed multiply operation in the multiplier; i.e., two numbers being multiplied are treated as 2s-complement numbers, and the result is a 32-bit 2s

complement number. The two registers associated with the multiplication process are the 16-bit temporary register (TREG0) which holds one of the multiplicands and the 32-bit product register (PREG) which holds the product. The other multiplicand in the operation is provided by the operand to the multiplication instruction. The result of the multiplication is stored in the product register. The product may then be stored by either transferring it to the accumulator or by storing the upper 16 bits or the lower 16 bits in data memory.

2.2.1.2 Parallel Logic Unit

Another unit within the CPU is the parallel logic unit (PLU). The PLU permits more efficient Boolean or bit manipulations, and it is shown in Figure 2.4. It is an independent unit which operates separately from the ALU. The PLU is used to directly set, clear, test, or toggle bits in a register or any data memory location.

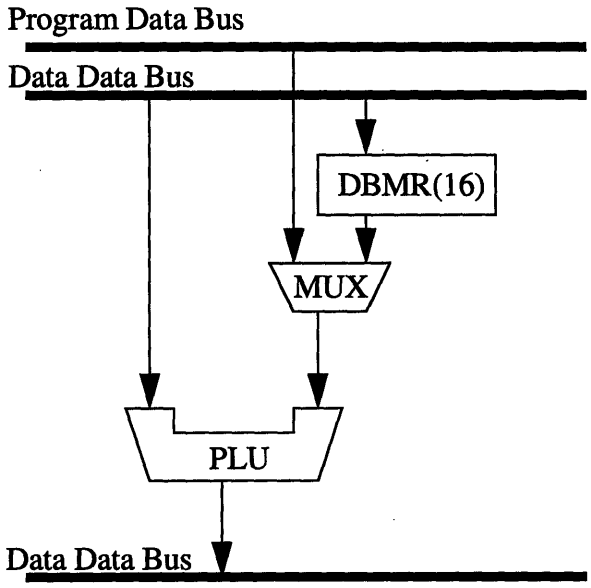


Figure 2.4 Parallel Logic Unit Block Diagram

The PLU provides a direct logic operation path to data memory values without affecting the contents of the accumulator or product register. The first operand is loaded into the PLU via the data bus. The second operand is accessed through either the program bus or the dynamic bit manipulation register (DBMR). Then, the PLU executes a logical operation defined by the instruction on the two operands. The PLU's results are written back to the same memory location from which the first operand was fetched.

The ALU and PLU can perform many of the same operations. The advantage of the PLU is that it can operate on data in memory without disturbing the contents of the arithmetic registers in the ALU. Hence, it is possible to perform arithmetic in the ALU and logical operations in the PLU without having to swap operands back and forth between memory

and the ALU registers.

2.2.1.3 Registers

The final primary element within the CPU is the registers. Twenty-eight core processor registers are mapped into the data memory space. These registers are listed in Table 2.1.

Table 2.1 Core Processor Memory-Mapped Registers

Name	Description
IMR	Interrupt Mask Register
GREG	Global Memory Allocation Register
IFR	Interrupt Flag Register
PMST	Processor Mode Status Register
RPTC	Repeat Counter Register
BRCR	Block Repeat Counter Register
PASR	Block Repeat Program Address Start Register
PAER	Block Repeat Program Address End Register
TREG0	Temporary Register for Multiplicand
TREG1	Temporary Register for Dynamic Shift Count
TREG2	Temporary Register for Dynamic Bit Test
DBMR	Dynamic Bit Manipulation Register
AR0	Auxiliary Register 0
AR1	Auxiliary Register 1
AR2	Auxiliary Register 2
AR3	Auxiliary Register 3
AR4	Auxiliary Register 4
AR5	Auxiliary Register 5
AR6	Auxiliary Register 6
AR7	Auxiliary Register 7
INDX	Index Register
ARCR	Auxiliary Register Compare Register
CBSR1	Circular Buffer 1 Start Address Register
CBER1	Circular Buffer 1 End Address Register
CBSR2	Circular Buffer 2 Start Address Register
CBER2	Circular Buffer 2 End Address Register
CBCR	Circular Buffer Control Register
BMAR	Block Move Address Register

Included in these registers is the register file that contains eight 16-bit auxiliary registers, AR0-AR7. The auxiliary registers may be used for indirect addressing of the data memory or for temporary data storage. Indirect auxiliary register addressing, which will be described later in this chapter, allows placement of the data memory address of an instruction operand into one of the auxiliary registers. These registers are pointed to by a three-bit auxiliary register pointer (ARP) that is loaded with a value from 0 through 7,

designating AR0 through AR7, respectively. The auxiliary registers and the ARP may be loaded from data memory, the accumulator, the product register, or by an immediate operand defined in the instruction. The contents of these registers may also be stored in data memory or used as inputs to the CALU.

The auxiliary register file is connected to the auxiliary register arithmetic unit (ARAU), as shown in Figure 2.5. The ARAU may autoindex the current auxiliary register while the data memory location is being addressed. Indexing by increments of one or more may be performed. As a result, accessing tables of information does not require the CALU for address manipulation; thus, the CALU is free for other operations in parallel.

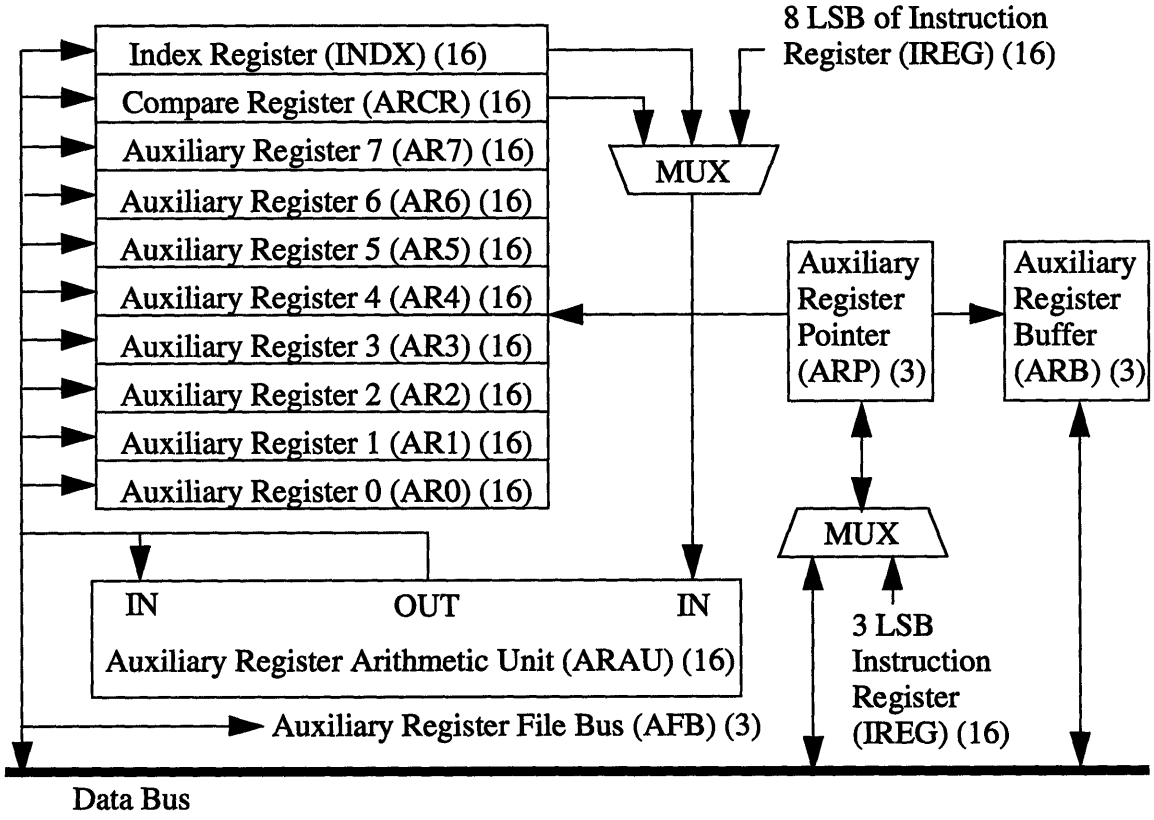


Figure 2.5 Auxiliary Register File Block Diagram

Along with the eight auxiliary registers, there are also the index register and the compare register. The index register (INDX) is one of the memory-mapped registers and is used to increment or decrement the address in steps larger than one. The auxiliary register compare register (ARCR) can be used as a limit to blocks of data or can be used as temporary storage if no comparisons need to be made. As is shown in Figure 2.5, the INDX register, ARCR register, and eight least significant bits of the instruction register can be used as an input to the ARAU. The other input is the current auxiliary register, which is being pointed to by the auxiliary register pointer. The output from the ARAU is

either a modified value in the auxiliary register or a modified status bit.

There are other registers such as circular buffer registers, control registers, and status registers. There are four circular buffer registers that are used for circular addressing. This type of addressing is useful especially in filters and performing convolutions. There are four primary status and control registers. These registers contain information about the current settings of the DSP and the configurations and settings chosen for the DSP.

2.2.2 Peripherals

The TMS320C50 peripherals are manipulated through registers mapped into the data memory space. These peripherals are accessed through a peripheral bus (TI bus). The peripherals include two serial ports, a timer, a software wait state generator, and 64K I/O ports. The serial ports can be configured as either a synchronous serial port or a time-division multiple-access port. They allow up to eight devices to communicate over the serial port. The timer provides a 16-bit countdown or event counter with a period and control register. The software wait state generators provide for interfacing to slower off-chip memory and I/O devices. This peripheral eliminates the need for external logic to create wait states. The circuitry consists of 16 wait generating circuits and is programmable to operate at 0, 1, 2, 3, or 7 wait states. Both program and data memory can be segmented into 16K words with one wait state generator per block. Finally, the 64K I/O ports are multiplexed with the data lines and are controlled via the I/O Space Select Signal.

2.2.3 The TMS320C50 Memory

Figure 2.6 shows the on-chip memory organization for the TMS320C50.

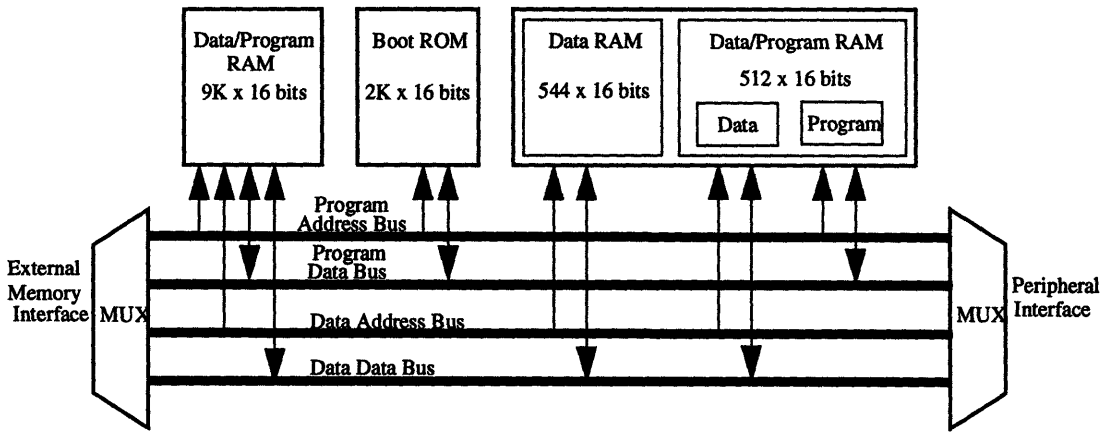


Figure 2.6 Memory Organization

The TMS320C50 has three on-chip memory blocks: 1K of dual-access RAM (DARAM), 9K of single-access RAM (SARAM), and 2K of program boot ROM. The SARAM can be mapped to program and/or data space and requires a full cycle to perform a read or a write. The DARAM is used as a data scratch pad by the CPU and can perform a read and

a write in the same cycle. The DARAM is configured in three blocks. The first block is 512 words and must be occupied in data space, and the second block is 32 words that also must be in data space. The third block is 512 words that can be either in data space or program space, depending on the user. Finally, the boot ROM is used for device tests, system initialization, test data, and as a boot loader.

In all, the TMS320C50 has a 224K memory range which is divided into four specific segments:

1. 64K program memory
2. 64K data memory
3. 32K global data memory
4. 64K I/O port memory

Of this memory, 192K can be off-chip. The program space contains the instructions to be executed as well as tables used in execution. The local data space stores data used by instructions. The global data space can share data with other processors within the system or can serve as additional data space. The I/O space interfaces to external memory-mapped peripherals and can also serve as extra data storage space. The memory maps for the program space and data space are shown below in Figure 2.7.

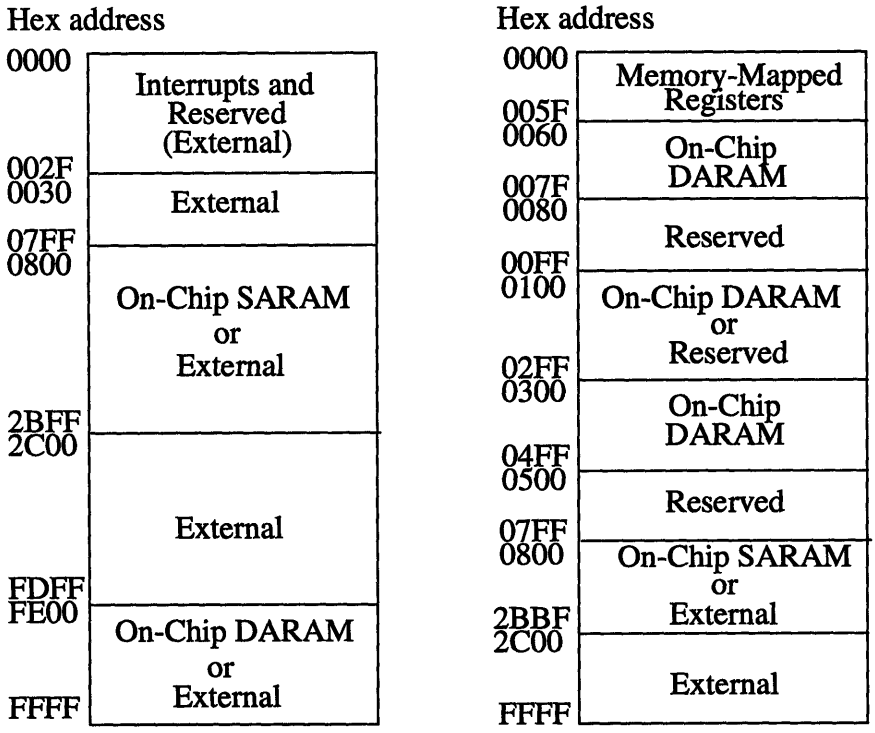


Figure 2.7 Memory Map of Program Space (left) and Data Space (Right)

2.2.3.1 Program Memory Space

The external memory program space can be used to address up to 64K 16-bit words. In addition, the TMS320C50 has on-chip ROM, single-access program/data RAM, and dual-access RAM. Software can configure these memory cells to reside inside or outside of the program address map. When they are mapped into program space, the device automatically accesses them when it addresses within their bounds. When the CALU generates an address outside these bounds, the device automatically generates an external access. The advantages of operating from on-chip memory are that there is a higher performance because no wait states are required for slower external memories, it has a lower cost than external memory, and it requires lower power. The advantage of external memory, though, is the ability to access a larger address space. Furthermore, when accessing external memory, the number of wait states can be determined by the user. This selection provides the user the freedom to select the number of wait states, if any, to allow for external data accesses. The program memory can reside both on- and off-chip, depending on certain status bits.

The program memory space contains the code for the applications. It can also be used to hold table information and immediate operands. The memory can be accessed only by the program bus, and the address for this bus is accessed only by the program counter (PC) when instructions and long immediate operands are accessed. The 'C5x devices fetch instructions by putting the PC on the bus and reading the appropriate location in memory. While the read is executing, the PC is incremented for the next fetch. If there is a program address discontinuity caused by perhaps a branch or call, the appropriate address is loaded into the PC. The PC is also loaded when operands are fetched from program memory. The operands are fetched from program memory when the device reads or writes to tables, when it transfers data to/from data space, or when it uses the program bus to fetch a second multiplicand.

2.2.3.2 Data Memory Space

The local data memory space on the 'C5x addresses up to 64K of 16-bit words. The on-chip space can be used as well, as data space if properly configured. When the on-chip memory is used, the device automatically accesses them when addressing within their bounds. When an address is generated outside these bounds, the device automatically generates an external access. The advantages to using on-chip memory are, similar to the case with program memory space, that there is higher performance because no wait-states are required, a lower cost and lower power consumption, but the additional advantage is that there is a higher performance because of better flow within the pipeline of the CALU. The advantage of operating from off-chip memory is, like in the program memory space case, the ability to access a larger data space. The 64K words of local data memory space include the memory-mapped registers for the device, which are located on data page 0.

2.2.3.3 Memory Addressing Modes

The local data space can be accessed through several different addressing methods. The first addressing method is called direct addressing. In direct addressing, the data address is formed by the concatenation of the nine bits of the data page pointer with the seven least

significant bits of the instruction word. This addressing scheme results in 512 pages of 128 words each, as shown below in Figure 2.8. The data page pointer points to one of 512 possible 128-word data memory pages, and the 7-bit address in the instruction points to the specific location within that data memory page.

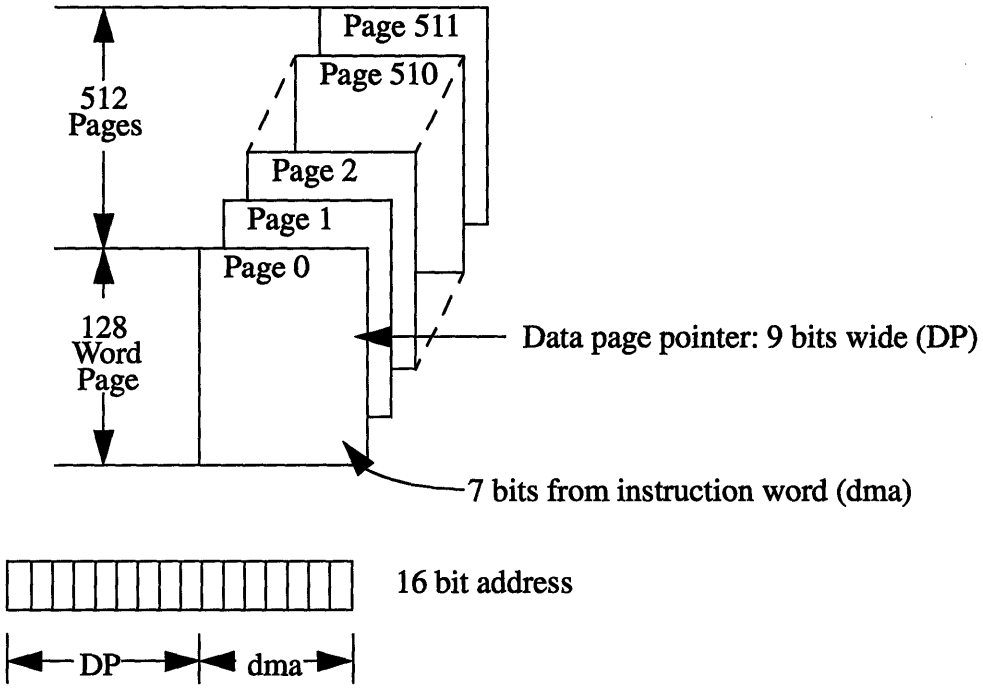


Figure 2.8 Example of Direct Addressing

Another type of addressing is indirect addressing. Indirect addressing is a powerful way of working with data in arrays. Unlike direct addressing, the address is not expressed in an instruction word, but is located in an auxiliary register. The use of the auxiliary register has the advantage that because it is a 16-bit register, it does not require paged addressing. The 16 bits can fully access any of the addresses in the 64K data memory space. More importantly, the auxiliary register may be automatically incremented by the ARAU after an operand is found; using this feature makes performing iterative processes on arrays of data very fast and easy. The INDX register may be used to increment the content of the auxiliary register by a step more than one. As shown in Figure 2.9, the contents of the auxiliary register pointed to by the auxiliary register pointer are used as the address of the data memory operand.

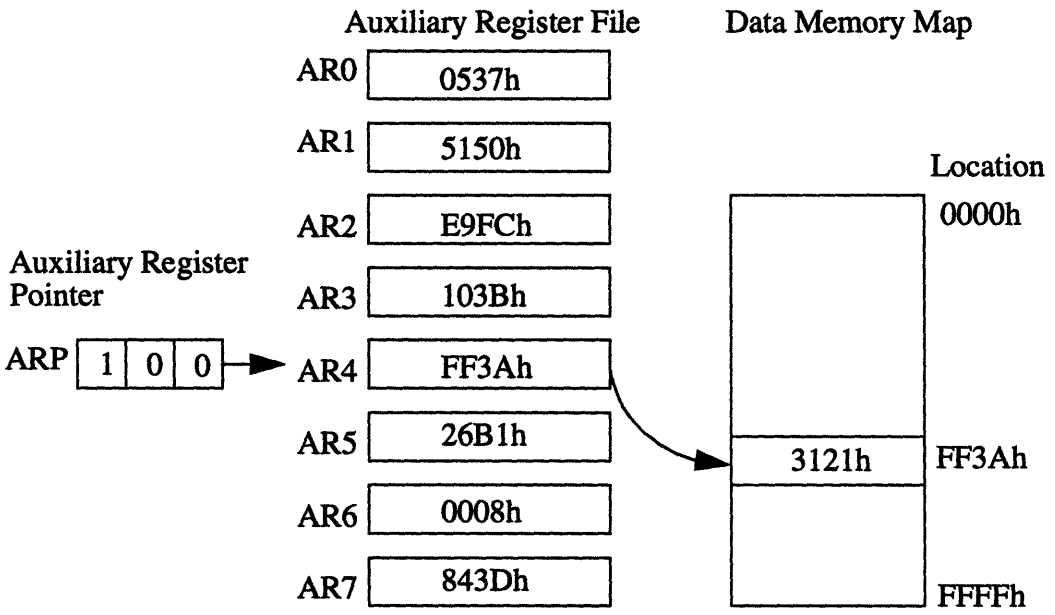


Figure 2.9 Example of Indirect Addressing

Immediate addressing is another type of addressing, in which the operand is part of the instruction machine code. In the case of short immediate addressing, the operand may vary from 1 to 13 bits. In long immediate addressing, the operand is contained in the word immediately following the instruction word. This is shown in Figure 2.10.

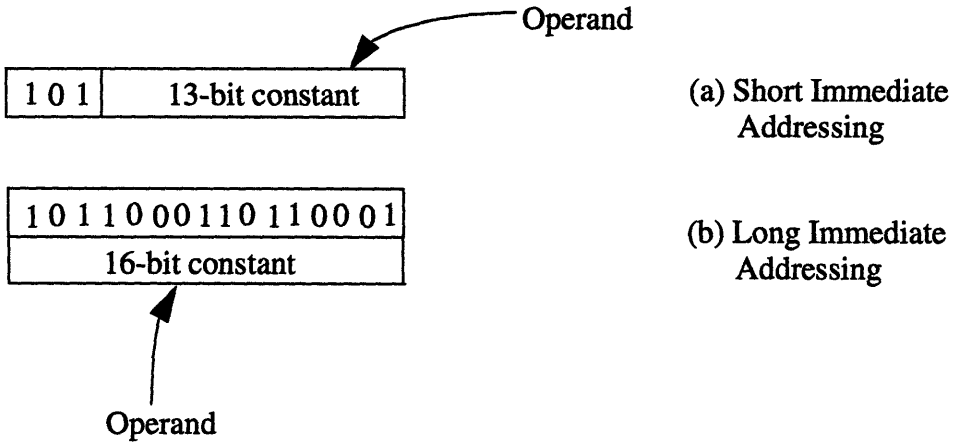


Figure 2.10 Example of Immediate Addressing

Other types of addressing include memory-mapped addressing and long direct addressing. Memory mapped addressing is very much like direct addressing except that in this case,

the data page pointer is always zero. Thus, the current data page is data page 0, which contains the memory mapped registers. This allows you to access the memory-mapped registers without the overhead of changing the data page pointer or auxiliary register. In long direct addressing, it is possible to specify an entire 16-bit address without the data page pointer.

2.2.3.4 Linker Command File

When actually placing code and data in the program space and data space, a linker command file is used. The linker command file allows the user to place the code and data in specific memory locations. A typical linker command file specifies the exact location of the code, the data, the stack, constants, and the heap. By writing a linker command file, the user has complete control of how to use the program space and the data space, and the user can determine whether to use on-chip or off-chip memory and how each should be used.

2.3 DSP Platform: Evaluation Module (EVM)

The platform containing the DSP chip that is used to run the hand printed character recognition algorithm is the evaluation module (EVM) [32]. This is a test platform that is typically used to test an algorithm and perform real-time applications. It allows full-speed verification of code and complete inspection of the DSP during its operation. On it, there is a TMS320C50 chip, which as mentioned above, contains 9K of on-chip single-access RAM and 1K of on-chip dual-access RAM. The external memory on the EVM, though, is different from that described above. Rather than having 64K of program space and 64K of data space as external memory, there is only 64K total external SRAM on board. This SRAM can be configured as program and/or data, depending on the user. When the memory is to be used as program space, the program memory map in Figure 2.7 must be followed, and similarly, when the memory is to be used as data space, the data memory map must be followed. Thus, for example, when using the memory as data space, the region from 0x0080 to 0x00FF is still reserved and cannot be used by the user. Recall that the memory is partitioned using a linker command file, described above.

The basic block diagram for the EVM is shown in Figure 2.11. The interconnects of the TMS320C50 include the target memory interface, host interface, I/O interface, analog interface, and emulation interface. Much of these interfaces were not used in this thesis but are included here for completeness. As mentioned, the TMS320C50 interfaces to 64K words of 25 ns SRAM, which allows zero-wait state memory accesses by the TMS320C50 operating at 25 ns instruction cycles. The PC-AT host interface is a 16-bit interface that supports the communication between the PC and the EVM by providing host I/O control and decode. The message host/target message interface provides a simple way to pass data between the EVM and the PC and maintain real-time operation. The EVM also includes a single-channel input/output voice-quality analog interface, which includes a single chip D/A and A/D conversion with 14 bits of dynamic range and a variable D/A and A/D sampling rate and filtering. Finally, the emulation interface allows emulation for the source code and loading of code on the TMS320C50.

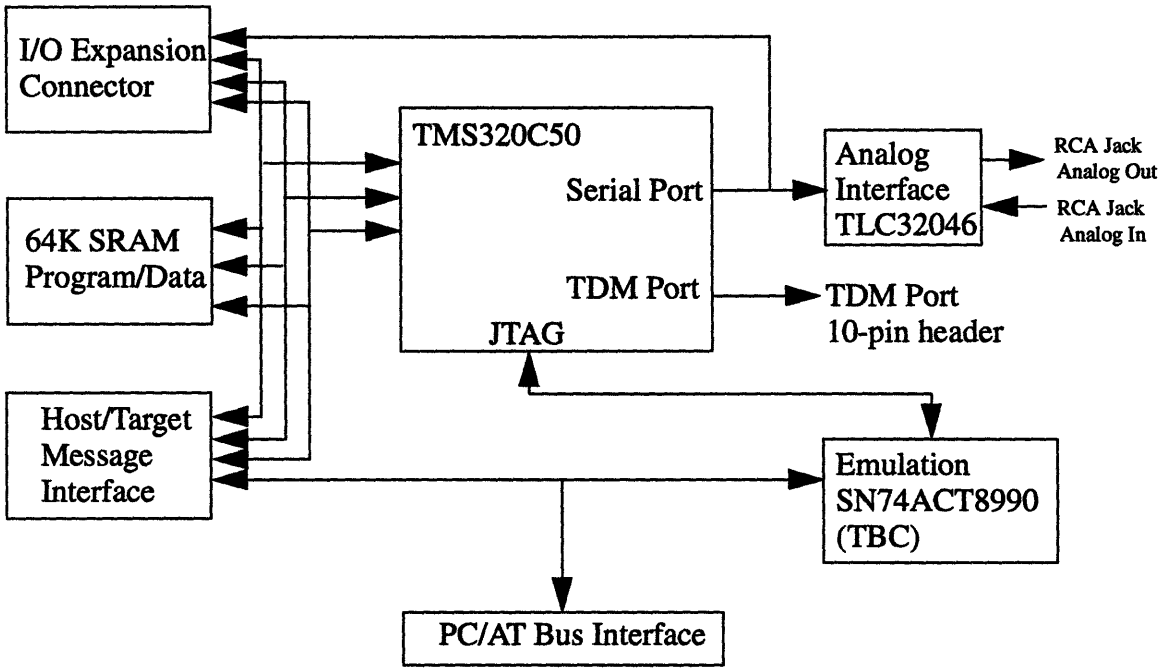


Figure 2.11 EVM Block Diagram

The EVM and PC must communicate when running applications. Typically, when an algorithm is ported to the EVM, the mathematically and data intensive code are run on the EVM, while the remaining functions such as graphics functions or I/O functions are left on the PC to run. This is because the DSP's strengths are in running highly mathematically and data intensive code, but this specialization prohibits it from being diverse enough to handle I/O functions and graphics functions. This provides a logical division of labor when running applications and necessitates the EVM/PC communication.

During the communication process, the EVM and PC send data back and forth in order to run the applications. The code on the EVM might require some data from the PC such as user input, and once the EVM is finished processing the data, the EVM must send the data back to the EVM in order to allow the PC to output the results to the user. Thus, data transfers are very critical in the communication process and must be coordinated between the two to prevent the two from being unsynchronized.

Typically, the EVM cannot send commands to the PC; only the PC may send the commands. This constraint simplifies the entire communication process, but it also imposes strict guidelines that must be followed in sending data and commands. This proper handshaking that is required between the PC and the EVM is extremely important, as a break in synchronization can cause the entire system to fail. For example, if a break in communication occurs where the PC is expecting data from the EVM while the EVM is

waiting for data from the PC, then the two are no longer synchronized and there is no way to recover. In a typical communication sequence, the PC sends a command to the EVM and the EVM responds to it. The command tells the EVM what to do; thus, the command might be requesting the EVM to send data to the PC, or it might be requesting that the EVM receive data from the PC. In either case, both the PC and the EVM must know exactly what data is to be transferred.

On both the PC and the EVM there is a control register which provides to the user system-level control and status information. There are two I/O port locations that are used to pass information between the PC and the EVM. One 16-bit bidirectional message register is mapped to both locations, and there are independent flags that allow the user to determine which port is being used. One port location is used for transferring data and dedicated to controlling data flags and the other port location is used to transfer commands and dedicated to controlling command flags. This method allows the same physical register to be used for both commands and data with independent control of the flags. On the PC side, the flags are the status bits TCMDINT and TDATAINT in the host control register which indicate whether the contents of the message register is a command or data. On the EVM control register, the flags are the status bits HCMDINT and HDATAINT, which indicate whether the contents of the message register is a command or data. There is also a control bit INTCLR\ in the EVM control register that can be used to clear HCMDINT, HDATAINT, TCMDINT, and TDATAINT.

Thus, in the communication process, the PC is the master and the EVM is the slave, meaning that the PC initiates communication by sending commands to the EVM, which simply responds to those commands. The communications protocol has three phases:

1. Command phase: PC initiates a command to the EVM. The EVM sets up to execute the command
2. Data phase: PC sends/receives data from EVM. EVM receives/sends data from/to PC
3. Cleanup phase: EVM clears outstanding command or data flags and gets ready to receive another command

A typical communication sequence when the PC is sending data to the EVM is shown in Figure 2.12. As shown in the figure, when the PC sends a command, the HCMDINT is set high to tell the EVM that a command has been written to the message register. The EVM then reads the command, which sets the TCMDINT bit and lets the PC know that the EVM has read the command. The PC then clears all the bits by sending a dummy command. The PC can then send the data one word at a time, each time waiting for the TDATAINT bit to be set high so that the PC can send the next word. Once all the data has been sent, the cleanup process involves the PC's sending a CLEAR command that will let the EVM know to toggle the INTCLR\ bit and clear all bits, thus preparing the EVM to receive the next command.

	HCMDINT	HDATAINT	TCMDINT	TDATAINT
PC sends command	1	0	0	0
EVM reads command and is ready for data	0	0	1	0
PC sends dummy command to clear HCMDINT, but this does not set TCMDINT	0	0	0	0
PC sends 16-bit data > repeat as	0	1	0	0
EVM reads data > required	0	0	0	1
PC sends CLEAR command that tells EVM that PC is finished	1	0	0	1
EVM reads CLEAR command	0	0	1	1
EVM toggles INTCLR\ bit to clear all bits and EVM is now ready for next command	0	0	0	0

Figure 2.12 Communication Sequence of PC sending data to EVM

The communication sequence when the EVM sends data to the PC is shown in Figure 2.13. In this case, much of the communication sequence is the same. After the PC sends the command, though, the PC must do a dummy read in order to set the HDATAINT bit high. The EVM waits for this bit to be high, and then it begins sending the data to the PC. After sending all of the necessary data, the EVM must do one dummy read in order to set the TDATAINT bit high; the PC can then begin the same cleanup process as in the communication sequence when the PC sends data to the EVM illustrated in Figure 2.12.

The sequences described above follow a strict handshaking protocol that does not allow the PC and EVM to become unsynchronized. As long as the PC and EVM know exactly which data is to be transferred and as long as they follow the above sequences, then the two will always be synchronized and work properly. Because only the PC can initiate commands, the EVM simply continuously polls the message register and checks the HCMDINT bit to wait for the next command. There are no timing problems, as the EVM can wait for any duration for the PC to be ready to send the next command.

The only initialization that must be done is that before any communication sequences begin, the INTCLR\ bit must be toggled high and then low. The communication protocol between the PC and the EVM is thus relatively simple and does not require much attention from the user once the process is begun. This attribute makes the EVM a very attractive platform in testing applications on a DSP chip.

	HCMDINT	HDATAINT	TCMDINT	TDATAINT
PC sends command	1	0	0	0
EVM reads command and is ready for data	0	0	1	0
PC sends dummy command to clear HCMDINT, but this does not set TCMDINT	0	0	0	0
PC reads dummy data	0	1	0	0
EVM sends 16-bit data > repeat as required	0	0	0	1
PC reads data > repeat as required	0	1	0	0
EVM does dummy read let PC know that it is done sending data	0	0	0	1
PC sends CLEAR command that tells EVM that PC is finished	1	0	0	1
EVM reads CLEAR command	0	0	1	1
EVM toggles INTCLR\ bit to clear all bits and EVM is now ready for next command	0	0	0	0

Figure 2.13 Communication Sequence of PC receiving data from EVM

2.4 Conclusion

The TMS320C50 is thus ideally suited for running real-time applications such as hand printed character recognition. Furthermore, with a platform such as the EVM that provides a relatively simple interface between the DSP and the PC, the TMS320C50 is thus well-suited for this algorithm implementation. This chapter provides a basic understanding of the DSP chip in general, and it can also serve as a reference when reading later chapters.

Chapter 3

Hand-Printed Character Recognition Algorithm

3.1 Overview of the Algorithm

The algorithm in this thesis is designed to recognize hand-printed alphanumeric characters in real-time, and it was devised by a research team working in Texas Instruments Italy. Printed characters refer to spaced, discrete characters, so each character must be completely separated from one another. Thus, no cursive script or run-on discrete characters are allowed. To enforce this constraint, the initial implementation of the algorithm requires that the characters must be written in boxes. This boxed, constrained environment is shown in Figure 3.1. Only one character may be written per box, and the character must be fully contained within that box. Note that each box is separated into two frames. As Figure 3.1 shows, the lower frames are reserved for characters with descenders such as 'g', 'j', 'p', 'q', and 'y'; the rest of the upper and lowercase letters may only be written in the upper frame. The boxed environment facilitates recognition because it forces characters to be separated and eliminates the need to perform character segmentation. As briefly mentioned in Chapter 1, the segmentation of handwritten characters is a nontrivial task that is under ongoing intense research. This constrained environment, equally importantly, serves three other primary purposes as well. It, first of all, helps to limit the characters' sizes and positions, which provides a way to maintain a certain degree of uniformity in the input. Second, it facilitates the recognition of upper and lowercase letters. Because there are some characters which, in upper and lowercase, are written with the same stroke sequence and whose difference is only in size, such as 'C' and 'c' or 'X' and 'x', then the boxes help to distinguish the two groups. In those cases, the relative size of the character to the box plays the key role in identifying the correct character. Finally, the boxed environment provides a division between those characters with and those characters without descenders. By initially checking whether or not the lower frame is empty, the number of comparisons can be reduced and the recognition accuracy will be improved.

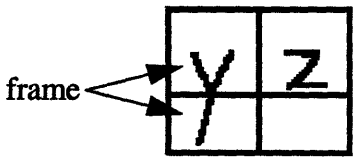


Figure 3.1 Boxed environment; Note that 'y' occupies both frames because it has a descender, while 'z' only occupies the upper frame

An alternative environment to that shown in Figure 3.1 could be one in which the box is divided into three frames. Thus, the middle frame could be reserved for only those characters with no ascender and no descender, such as 'x', while the upper frame would be

for characters with ascenders and the lower frame for characters with descenders. This type of environment would easily eliminate any of the confusion mentioned above with the recognition of a 'C' and 'c', for example.

Because the algorithm is implemented in real-time, there is temporal information that can be taken advantage of and used to improve recognition. One such data is the stroke information. When a printed character is written, the character can be broken up into strokes. A stroke can be defined as a collection of points that are written from pen-down to pen-up. Thus, as long as the pen does not leave the writing surface, it is considered one stroke. Figure 3.2 illustrates how a character such as 'E' can be written with one, two, three, and four strokes [8]. The knowledge of the number of strokes and the direction of the stroke enables the algorithm to achieve a high recognition accuracy.

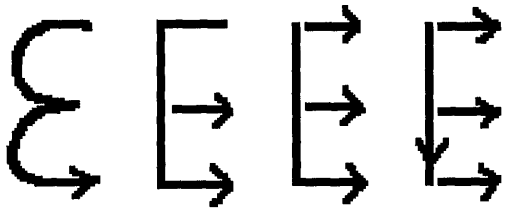


Figure 3.2 From left to right, the number of strokes that it takes to write an 'E' increases from 1 to 4.

Another aspect of this algorithm that aids in improving recognition is that it allows users to teach the recognizer to recognize a specific writing style. This recognition system can be user-independent, meaning that no learning is required, but recognition results improve significantly with user training. User training involves writing samples of the characters to be recognized; the characters are then sent to the algorithm to be processed and the appropriate data are stored in the symbol database. The symbol database is the dictionary that will be used to identify written characters, and it is created and contains each symbol that will be recognized along with its appropriate data. That data will be used for future recognition by comparing the processed data from the input to the processed data of these writing samples in the symbol database. This user training provides more flexibility by allowing multiple users to train the recognizer to recognize their individual writing style.

Once a particular writing style is learned, a user can write characters to be recognized. If a misrecognition should occur or if the character is not identified at all, the algorithm allows the character and the correct identifier to be inserted into the symbol database. This insertion capability allows for a higher level of flexibility in user-training as poorly written characters can be accidentally written during the learning stage and entered into the symbol database, and it eliminates repeated mistakes. To avoid future misrecognition and to eliminate characters that degrade the system by causing recognition errors, the algorithm checks the number of times a character has caused a recognition error. If that character in the symbol database has reached a number greater than a certain threshold, the

character is eliminated from the database altogether.

The basic flow diagram of the process of recognizing a single character is shown in Figure 3.3. When a character is written, the data that is collected is the coordinates of the points making up each stroke in the character. That point data is sent to the preprocessing phase. Preprocessing is done initially to take the printed input character and clean and normalize it. This step is necessary because of the imperfections and the nonuniformity of hand printed characters. Common imperfections include size variation, accidental pen lifts during writing, and uneven point distribution among a stroke. The preprocessing consists of scaling the character, interpolating and smoothing the character, and thinning. After the preprocessing, it is ready to be sent to the next phase, the topological engine. The topological engine takes the cleaned input from the preprocessing and attempts to accurately and faithfully describe the features of the character while at the same time minimizing the amount of data necessary to accomplish this description. The character's features are obtained along with the temporal information that is a result of real-time recognition. After all the appropriate processed data is obtained, the character is finally ready for the recognition phase. The recognition is performed by comparing the topological engine data to the data from the symbol database. Recall that the data from the symbol database was obtained by sending the learned writing samples to the preprocessing and topological engine phases. The character in the symbol database with the data closes to the input character's is used as the recognized identifier.

The next sections will describe the three phases that were briefly mentioned above in much more greater detail. The first section will discuss the preprocessing phase.

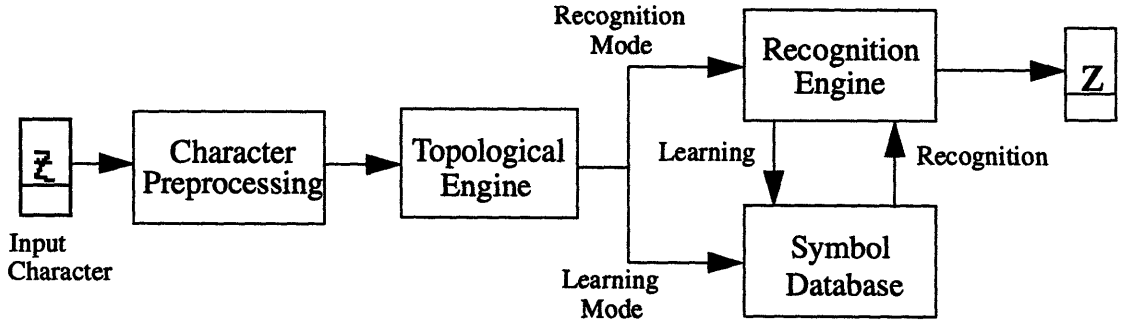


Figure 3.3 Overall Algorithm Flow Diagram

3.2 Preprocessing Functions

As mentioned above, the first step in recognizing a character is preprocessing. The input to the preprocessor is the coordinate data from each point in each stroke of the character. This point data then provides both spatial and temporal information, for the points are stored in the order they were written. Figure 3.4 illustrates an example of the point data for a hand printed 'A' that will be the input to the preprocessing.

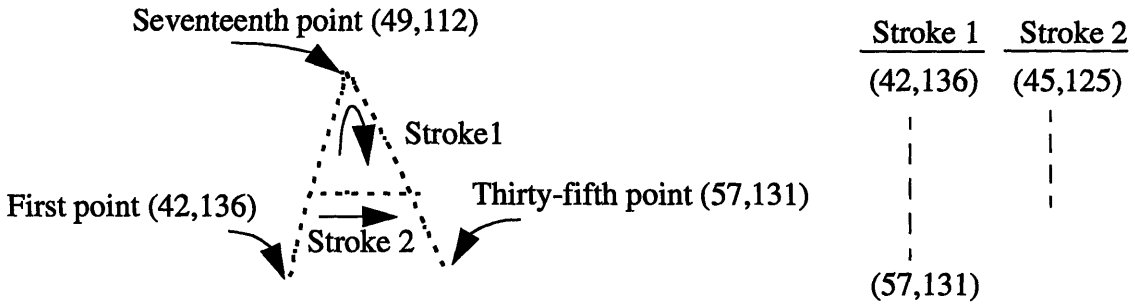


Figure 3.4 Point data for a hand printed 'A'

Because the input can vary so widely, the preprocessing is a very essential step. The purpose of the preprocessing is twofold. It first serves as a chance to convert the data to a format that will facilitate recognition, and it second serves as a way to clean the input. All characters as a result of preprocessing are converted to an NxN character matrix. This input standardization enables a more accurate comparison of topological data between the input character and the character from the symbol database Input needs to be cleaned because characters can vary in size, and furthermore the distribution of points can be very different. For example, if the character is written slowly, the number of points is very high. The slow pen movement enables the tablet to sample the pen's location numerous times and provides many points in the character. A fast pen movement can greatly reduce the number of points in a character. This is because rapid movement does not allow the tablet to sample many points, and the result is very few points representing a character. This wide distribution of number of points makes preprocessing a necessary step to eliminate points if there is slow pen movement and add points if there is fast pen movement. Finally, the preprocessing can eliminate redundant information that does not add to the character's identity, and this reduction of information will speed up the recognition process and the topological engine process. Often times in a character, there is too much information, and not all of the information is independent of one another. So, the preprocessing can help to trim the information and include only the necessary information to use for recognition.

The first step in preprocessing is scaling the points. As mentioned before, characters are generally written in varying sizes, so there is a need to normalize all the characters and eliminate the variations cause by size differentials. The rest of scaling is the transfer of the points of the stroke to an NxN square character matrix. Scaling is accomplished by multiplying each point by a scale factor. This multiplication has the effect of expanding the character in all directions by the scale factor. Then, each point is fit into a corresponding relative location in the character matrix through a formula that normalizes the character's size. Thus, each character, no matter how large or small it is written, is scaled so that it fits in the square character matrix. Scaling and the next step, interpolation, are closely linked and are generally performed simultaneously.

Interpolation is performed for two reasons. It first eliminates redundant information that

is caused by slow pen movements and second adds points if the points are too far apart due to fast pen movement. In the interpolation phase, at the same time as scaling, the points are put in the NxN character matrix one at a time. Each point in the character stroke corresponds to a location on the character matrix. At these matrix locations, a '1' is placed, representing a black pixel. When there is no point in the character stroke that corresponds to a matrix location, a '0' or a white pixel, is placed. The points are processed consecutively in the same order that they were written. If a point in the stroke corresponds to matrix location that already was '1' because it belonged to previous point in the stroke, this implies that the two points were too close together and were redundant, so the second point is discarded. This is the eliminating of redundant points. On the other hand, if two consecutive points are in such a way that they don't correspond to neighboring locations in the matrix, then a linear interpolation is formed to fill the gap between them. This is shown in Figure 3.5. In the figure, matrix locations (i+1,j) and (i+1,j+2) correspond to two consecutive points in the character stroke. Thus, the points were far apart due to fast pen movement, so they did not correspond to consecutive matrix locations. The interpolation routine fills in the matrix at (i+1,j+1) to close the gap and smooth the character matrix. The figure shows the character matrix for the letter 'A" after scaling and interpolation.

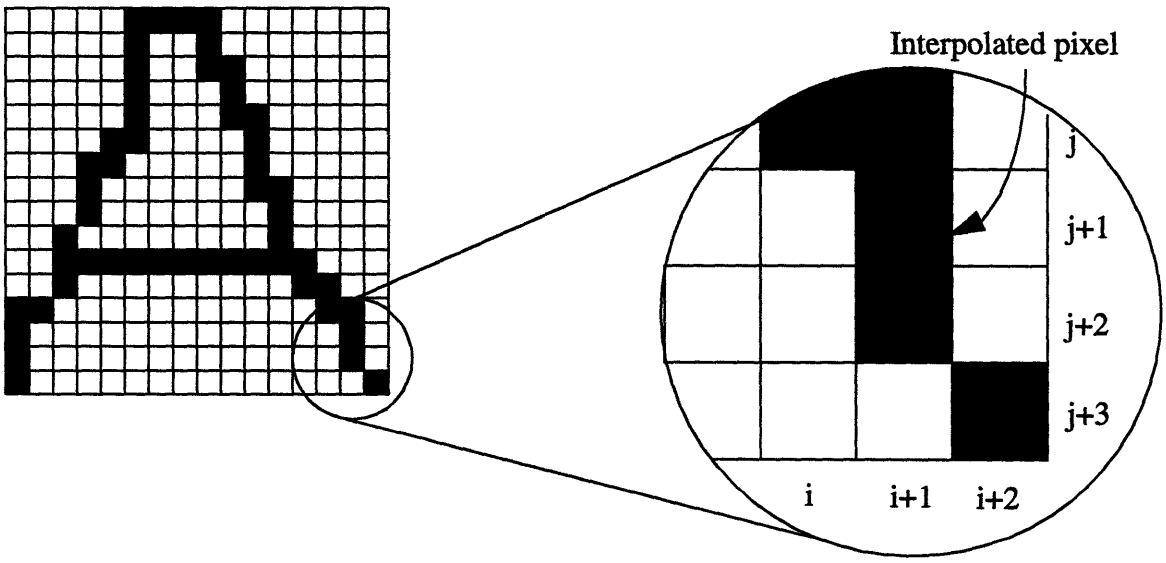


Figure 3.5 Example of interpolation. Pixel at (i+1,j+1) is result of interpolation.

After scaling and interpolation, the final step in preprocessing is thinning. The purpose of thinning is to eliminate redundant points and information. The thinning process takes the character matrix as input and it looks only at those matrix locations that contain a '1' and thus correspond to a point in the character stroke. It then eliminates those points in the matrix that are considered extraneous. This is best shown in Figure 3.6. The 2nd point in Figure 3.6 as shown is eliminated because the point does not add anything in terms of topological information or recognition information; i.e., it is redundant. Only the 1st and 3rd points, as shown in Figure 3.6, remain after thinning.

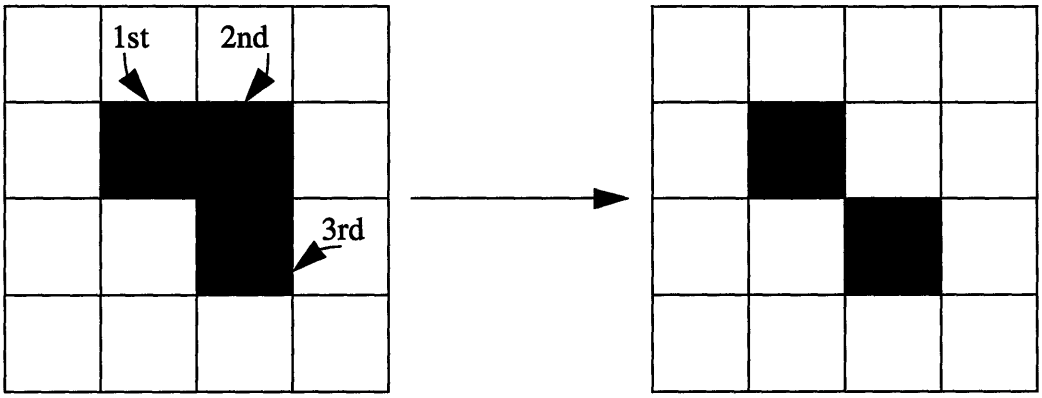


Figure 3.6 Result of thinning. 2nd point is eliminated.

Thus, thinning eliminates corners that do not contribute to a character's identity. Figure 3.7 shows the result of taking Figure 3.5 as input and thinning it, and it is the final result from preprocessing and is the input to the topological engine.

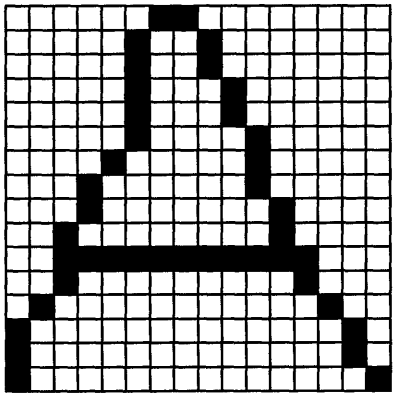


Figure 3.7 Final result after all of preprocessing

3.3 Topological Engine

In the topological engine phase, the goal is to provide a faithful description of the character and its features while at the same time minimizing the amount of data necessary to do so. The information that is obtained includes the character height and width, the features, the microfeatures, and the dynamic information. The first step is to obtain the height and width of the character. This step is accomplished by examining the point data of the character and creating a character box. The minimum x- and -y values from all the points are used as one corner of the character box, and the maximum x- and y- values from all the points are used as the opposite corner. The box's dimensions are then the character's height and width. Each value is stored in one byte.

The next step is to perform feature extraction. Feature extraction is the step that is used to accurately depict the primary features of a character with only a minimum of data. The way it is accomplished is comparing the NxN input character matrix with M feature maps. These feature maps were determined after extensive simulations, and they are simply NxN matrices, each with N matrix locations that are filled with a '1'. The input character matrix is simply compared with each feature map by performing a simple bitwise AND operation for all N*N corresponding matrix locations. The actual optimized implementation is different from this, but this is the basic principle. The number of matrix locations that are in common between each map and the input character matrix is recorded, and then all M of these values are placed through a threshold formula, similar to the formula below. The threshold value is stored in the minimum number of bits possible, and the total storage required for the feature extraction data for one input character is 20 bytes. The formula below illustrates the thresholding that occurs in the feature extraction process. Let numpoints be the number of points that a feature map and the input character matrix have in common, and let δ_i be the threshold values and K be the number of thresholds.

$$\text{if } numpoints \leq \delta_i, \text{ then } score = i \qquad \text{for i=0...K}$$

Figure 3.8 illustrates an input character matrix being compared with a feature map. The figure is not an actual representation of what occurs in the algorithm because the algorithm is considered proprietary information by Texas Instruments, but it serves as a good illustration of how the general feature extraction process operates.

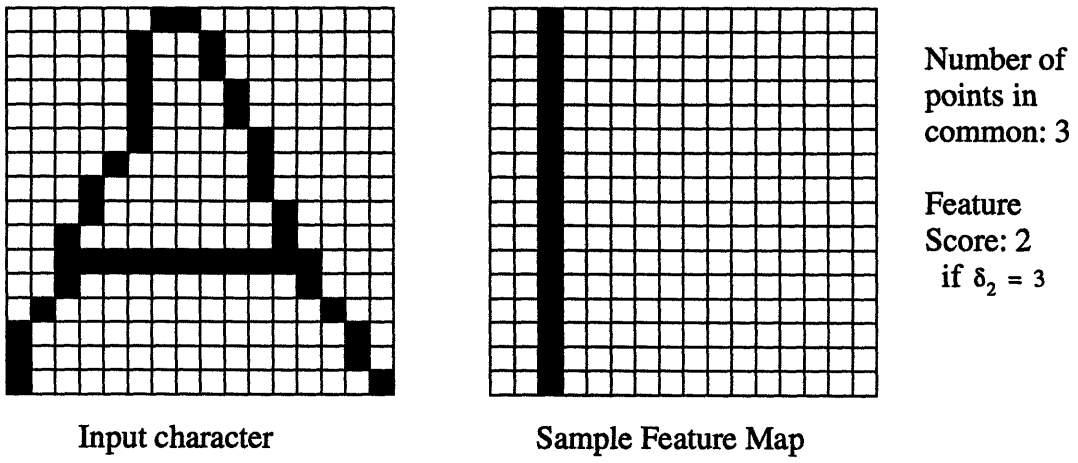


Figure 3.8 Illustration of feature extraction

The next information that is obtained in the topological engine is the microfeatures. This data is compiled to gain information and identify the relative position of neighboring points. This data is determined in the following manner. The first L rows of the NxN

character matrix are used to collect microfeature detection data, and it starts from the top and goes downwards. The first row is examined, and the first P locations with a '1' in them, implying that location corresponds to a point in the character stroke, are used. If there are not P matrix locations on that row with a '1' in them, then for however many there are, they are used for the microfeature detection data. The T neighboring locations on the row below are examined. For each location that has a '1' in it, a score is given. The score for all T locations are summed up, and this is the microfeature detection score. This process is illustrated in Figure 3.9. Again, because this algorithm is considered by Texas Instruments to be proprietary information, this illustration is not an accurate depiction of what actually occurs in the algorithm, but it still provides a good understanding of how the microfeature detection works. In this example, T is given to be 5. Thus, in Figure 3.9, the location at (j,i) is the location being examined because it has a '1' in it, meaning that it contains a black pixel, and it is within the first P points in the row with a '1'. The five neighboring locations on the row below are (j-2,i+1), (j-1,i+1), (j,i+1), (j+1,i+1) and (j+2,i+1). Each has a score associated with it that is added if it contains a '1'. Of the five neighboring points to (j,i), the ones at (j-1,i+1) and (j,i+1) have '1's in them, so the microfeature detection score is Score2+Score3.

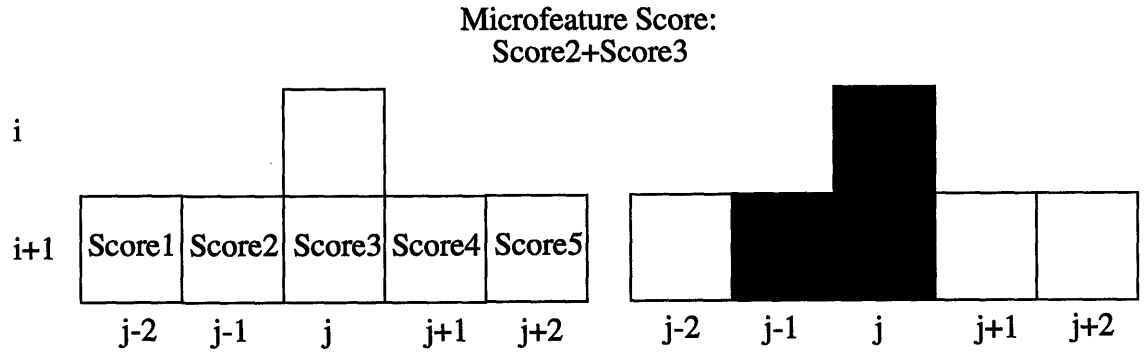


Figure 3.9 Illustration of Microfeature Detection

This microfeature detection is performed for the first P or less locations with a '1' in them and is done for the first L rows. Thus, there is a maximum possibility of L*P calculations, and each microfeature detection score is stored in the fewest possible bits. A total of 29 bytes of storage are required for the microfeatures.

The next information that is obtained is the dynamic information, which is available because of the ability to watch the characters as they are written. Recall that the point data is temporal as well as spatial, as the points are stored in the order they were written. The purpose of the dynamic information is to provide additional insight in determining the characteristics of a character and to facilitate recognition. The dynamic information is the relative position of points to the first and last points and to neighboring points. The information that is obtained is three sets of Q data points. The number Q was obtained

from extensive simulations. The three sets of data are the relative position of the x-values of the points to the first and last points, the relative position of the y-values of the points to the first and last points, and the relative position of points to each other.

The points of a character are taken, and the Q points used for the dynamic information are taken so that they represent points that are almost equally spaced among the character length. The points that are used are then compared to the first point, to the last point, and to the previous point. This step provides information on the points' locations in a temporal manner as well as a spatial one.

The following figure illustrates how points are compared to the first and last points of the character. In this Figure 3.10, a box is constructed using the first and last points of the written character as opposite corners of the box. Then, each of the Q points that are examined for the dynamic information is compared to this box, looking at the x-values first and then the y-values. A score is calculated for each of these Q points by determining the position of the point in relation to the box. For the x-value calculation, if the point is to the left of the box, then the dynamic score, as shown in the figure, is A. If the point is to the right of the box, then the dynamic score is B. Otherwise, the point lies either within the box's x-coordinates. In this case, it needs to be determined which corner of the box is the first point and which corner of the box is the last point, and depending on this information, the dynamic score is either C or D. Similarly, for the y-values, if the point is above the box then the score is A, if the point is below the box then the score is B, and otherwise, the score is C or D depending on which corners are the first and last points. These x- and y-value calculations are performed for the Q points until they all have been compared to the first and last points of the written character.

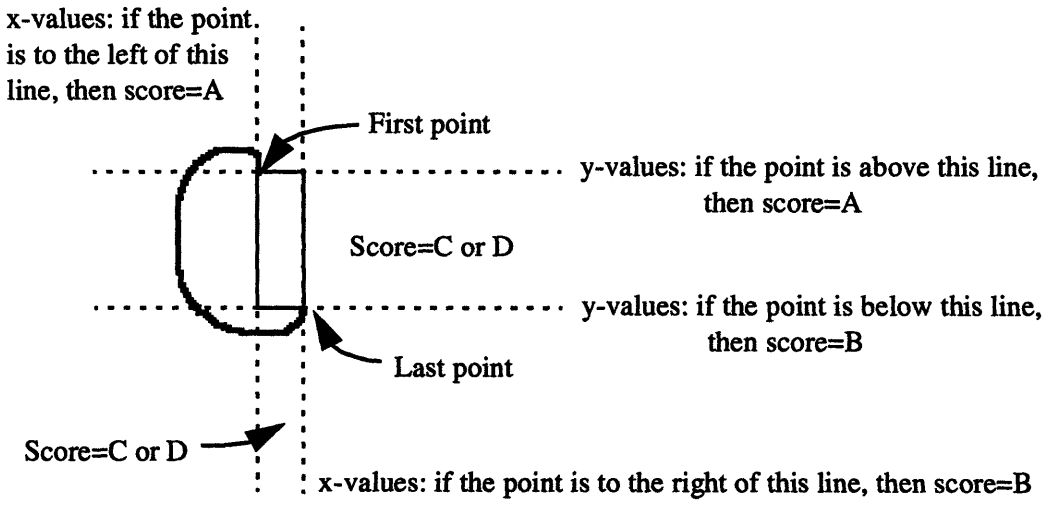


Figure 3.10 Illustration of obtaining x- and y- value dynamic information from first and last points of the written character

The other set of dynamic information that is extracted from the temporal data is the comparison of points that are temporally next to each other. In this case, two neighboring points, of the Q points, are examined at one time. Of the two points, the point that was first written is used as the origin, and a line is drawn from this point to its neighboring point, which was written at a time later than this earlier point. The slope of this line is calculated, and the dynamic score given for these two neighboring points is determined based on this slope, whether positive, negative, zero, or infinite, and on the quadrant in which the line is located. This process continues for all Q points.

The three sets of dynamic information are then stored in the minimum number of bits possible, and all of the dynamic information obtained from the Q points are stored in 27 bytes. The next section will now describe how this topological information is used in the actual recognition process.

3.4 Recognition Functions

The recognition phase takes the data from the topological engine phase and uses that to select the correct recognized identifier for the input character. It compares the input character to all of the characters in the symbol database to arrive at its decision. However, this algorithm first performs a check to see if the input character is a special punctuation mark. These special punctuation marks, which include only the comma, semicolon, colon, and period, are not learned by the recognizer; that is, there is no user training required to recognize those four punctuation marks. Instead, as part of the algorithm, these punctuation marks have been modelled and their features extracted for recognition. The remaining punctuation marks can be recognized by this algorithm, but user training is first required.

Thus, the check that the recognizer performs on an input character to determine whether or not it is one of the four punctuation marks mentioned above is to examine the character's height and width. If the height and width are below some punctuation threshold, then the input character is examined more closely to accurately determine the character's correct identifier. The recognizer examines the input's topological data to determine an answer; for example, if the number of strokes taken to write the character is two, then the character could only be a semicolon or colon. If the number of strokes is one, then the character could only be a comma or period. Using a formula that takes the topological data as input, it is determined whether or not the character is a punctuation mark.

If it is not a punctuation mark, then the input character begins the true recognition phase. The general flow of this phase is shown in Figure 3.11. The input character is compared with a character from the symbol database one at a time. The first check is to see whether or not the lower frame is empty. This reduces the number of comparisons necessary between the input character and the characters in the database. If the input character and the character in the database have different frames, then the comparison is not made and the next character in the database is used for the next comparison. The input character is

compared in each category and a score is given based on the degree of similarity. This score is weighted according to the importance of that category. At the end of all the comparisons, the character in the database with the lowest score is the recognized character.

The characters are compared in the number of strokes it takes to write the character. This category is particularly important in weight in the user-dependent mode, as the user is likely to write the same character the same way each time. And even if the user writes a character different ways, each way can be inserted in to the database to be compared with the input character.

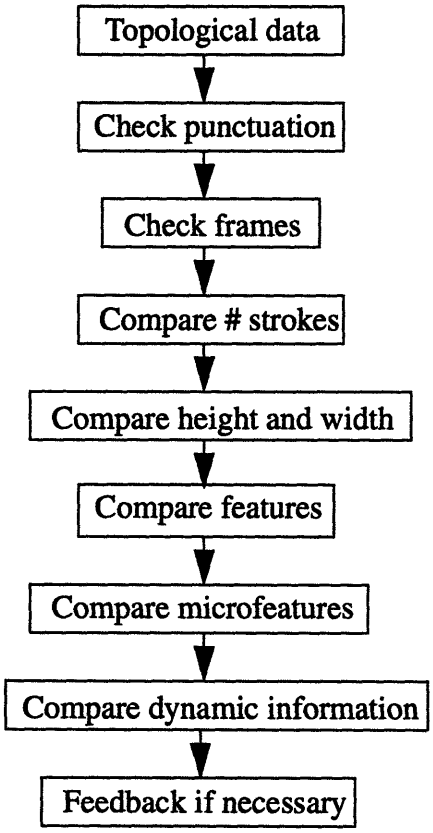


Figure 3.11 Recognition Phase

After the number of strokes is compared, the next step is to compare the height and width of the characters. This is important to distinguish lower case characters from upper case characters. Again, this result is greatly aided by the fact that the characters must be written in a box. The box facilitates the user's decision of how high to make capital letters. It also sets a limit to the width of the character. The height and width are compared and weighted and a score is calculated. The greater the difference, the higher the score. The height and width should not be emphasized; i.e., the weight should not be great if the

characters are in unboxed environment, and even in the boxed environment, not too much weight should be placed on them initially because of the user variance in character writing size. This algorithm has a special feedback routine feature that takes into account characters whose uppercase letter and lowercase letter are similar in stroke sequence and differ only in height. This feedback routine will be discussed later.

After the height and width, the features are compared. For each feature map, there is a score that reflects the number of points in common between the feature map and the character. For the input character and the character in the database, the feature scores are compared for each feature map and the differences between the two characters' scores are summed. The score is then weighted according to the importance of the features, and the higher the score, the more different the two characters are. Features are very important in alphanumeric characters and are weighted fairly heavily for this algorithm.

After the features are compared, the microfeatures are compared. Recall that the microfeatures are scored by examining the first 5 matrix locations that have a '1' in them, and thus represent black pixels, in each row and examining the three neighboring locations in the row below. The score is calculated based on the number of black locations below it. The recognition score for the microfeatures is obtained by looking at each of the fifteen rows, one at a time. Within each row, each black location is examined separately and compared between the input character and the character from the database. The score is determined by giving a higher score when the two characters' microfeatures are more different. For instance, if the two black locations of the two characters do not have similar neighboring locations, i.e., the neighboring locations below have a different pattern of locations with a '1' in them, the score is high. If the characters' two black locations have all the same neighboring black locations below, then the score is low; otherwise, the score is someplace in between. The microfeatures are very important because they, too, help create a feature map understanding of the characters.

Finally, the last comparison that is made is the dynamic information. The dynamic information for both the input character and the reference character are examined; the data comparing the points to the first and last points are examined first. Depending on whether the two characters have similar data, the scores are obtained from a certain set of matrices. Basically, the goal is that if the dynamic information is very similar, then the score will be low; else, if the dynamic info is very different, the score will be high. This information for the algorithm is weighted very heavily. This is because this is a real-time recognition algorithm, and the writing of alphanumeric characters often is the same pattern. Character sets like Kanji, where the dynamic information is perhaps even more important, can place an even greater weight on this category.

After the recognition, the reference character with the lowest score is considered the recognized identifier. Before returning this as the exact recognized symbol, though, the algorithm checks if the identifier is among those characters in Table 3.1. If it is, then the feedback is required. Table 3.1 contains those character pairs whose recognition is more difficult because the characters are written with similar stroke sequences but differ only in

size. Thus, feedback allows the algorithm to re-examine the input character and this time place a much higher weight on the character height and width. After the feedback, the result is the recognized identifier of the input character.

Table 3.1: Feedback characters

C	c
M	m
N	n
O	o
S	s
U	u
V	v
W	w
X	x
Z	z
l	e
h	n
/	\
1	\

3.5 Conclusion

This algorithm, which recognizes hand printed alphanumeric characters, has as its strength the fact that it is compact, fast, and efficient, yet it is able to achieve a high recognition accuracy. Furthermore, with the improvements described in the following chapters, this algorithm is ideal for commercial applications such as pen computers and personal digital assistants. The next chapters will discuss these issues in greater detail.

Chapter 4

Hand Printed Character Recognition Implementation

4.1 Setup

The hand printed character recognition algorithm in this thesis is implemented for real-time operation on a DSP chip; i.e., the characters are processed and recognized on the chip as they are written. Figure 4.1 shows the basic setup used. The PC is a 486 33MHz computer, and inside is the EVM that contains the Texas Instruments TMS320C50 DSP chip, which is described in Chapter 2. The characters are written on an electromagnetic digitizing tablet using a pen-like writing instrument, the stylus, and this writing process is analogous to that when using a pen and paper.

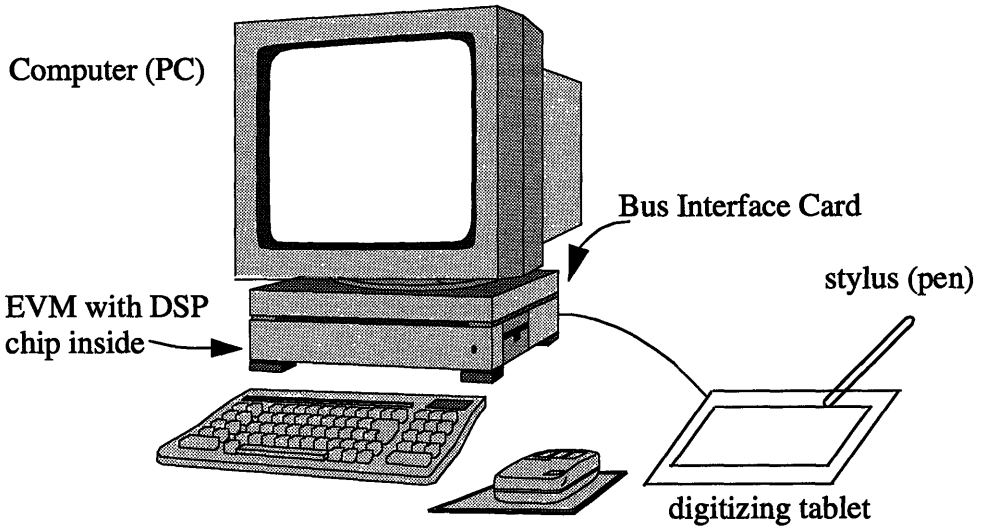


Figure 4.1 Setup of Hand Printed Character Recognition Implementation

The stylus is shown in more detail in Figure 4.2. It is cordless, and it has two switches--the tip switch and the barrel switch. The barrel switch turns on the stylus when pressed, and it allows the stylus to be activated even when it is not in contact with the tablet. The tip switch is pressure-sensitive and turns on the stylus only when the stylus tip is pressed down against the tablet. When the stylus is activated, strokes can be written, and they will be recorded by the tablet.

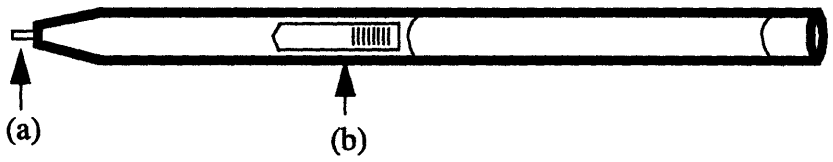


Figure 4.2 The cordless stylus. (a) tip switch (b) barrel switch

The electromagnetic digitizing tablet used in this thesis is Wacom's PL100V; it has a 640x480 LCD screen with sixteen shades of grey. The tablet has two modes: a transmit mode and a receive mode, and it switches between the two every twenty microseconds. During the transmit stage, the digitizing tablet sends a signal that produces electromagnetic resonance in the stylus. The stylus stores this energy using a coil and capacitor resonant circuit. Then, during the receive stage, the stylus sends a signal to the tablet that tells the tablet the amount of pressure being applied on the tablet's screen by the stylus and the type of switch being used by the stylus. By measuring the signal strength from the grid wires below the screen, the tablet can then compute the stylus's location and send these coordinates to the bus interface card in the PC that is shown in Figure 4.1.

The bus card is the input/output interface and serves as the VGA video card and serial port. The card allows the video output from the computer to be simultaneously displayed on both the PC's VGA screen and the tablet's LCD VGA screen, and the card also communicates with the tablet through its serial port. Thus, once the tablet sends the coordinate data to this bus card, the card then sends instructions to the tablet that tells the LCD screen what information to display.

4.2 HPCR User Interface

The user interface for this recognition algorithm is shown in Figure 4.3. There are two writing zones: the single character writing zone and the drawing zone. The single character writing zone allows up to twenty-four characters to be written in the boxed environment and then recognized by the algorithm. The drawing zone allows figures, symbols, and text to be drawn without being sent to the recognition algorithm. This option is useful when the user wishes to write or draw symbols that do not need to be recognized, and it is essential in applications such as the personal digital assistants mentioned in Chapter 1. In that type of application, recognition is not always necessary, but instead the electronic ink, which is the strokes as they appear on the tablet's screen, is adequate.

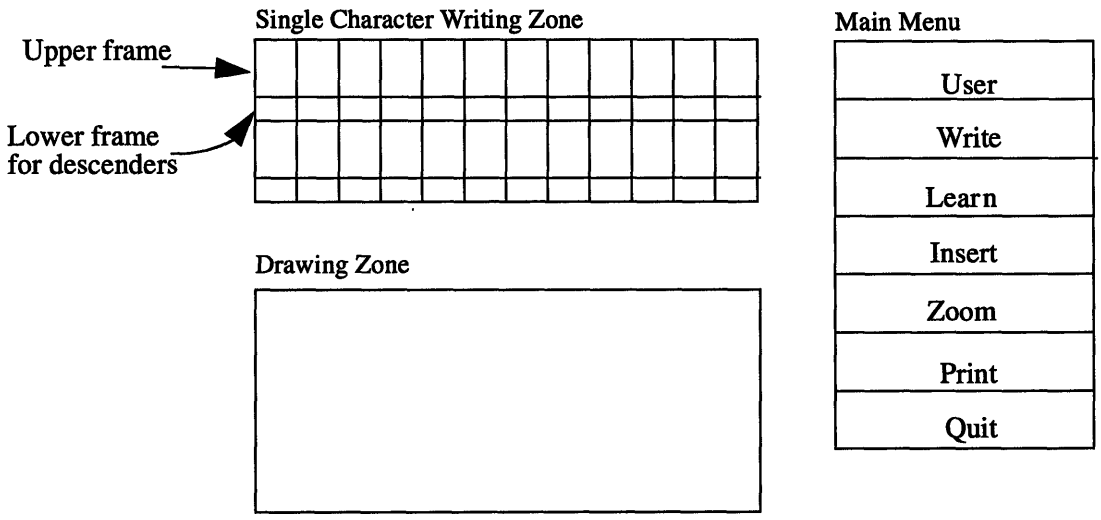


Figure 4.3 HPCR User Interface: Main Menu

As shown, there are seven options in the main menu. The USER option allows the user to select an appropriate database to use as the reference dataset for recognition. This flexibility permits multiple users to use the recognition system and enables personal writing styles to be taken into consideration, which increases recognition accuracy. When the option is selected, the user is prompted to enter a username using a QWERTY keyboard. The user database, if it exists, is then loaded up and used as the symbol database for recognition.

The WRITE option allows the user to write in either writing zone. When the user selects the WRITE option by depressing the stylus on this option, the user can then write in either the single character writing zone or the drawing zone. The user may write as much or as little as desired. The writing menu is shown in Figure 4.3. The user can, at any time, during the writing process, select one of the four options in the writing menu. The BACK option removes the last stroke written by the user, the CLEAR option clears all the strokes written, the DELETE option allows the user to select one of the single character boxes to be deleted and thus allows the user to rewrite a single character, and the RECOGNIZE option begins the recognition of all the characters written in the single character writing zone.

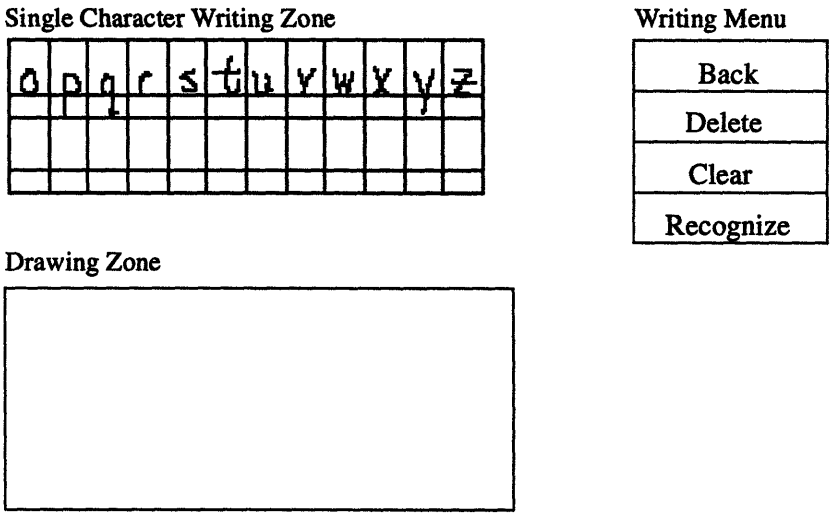


Figure 4.4 HPCR User Interface: Writing Menu

The LEARN option begins the learning session. When the user selects the LEARN option, the screen is similar to that shown in Figure 4.4. Twelve characters from the symbol database appear on the screen at one time, and the user writes the characters to be learned in the appropriate box. In order to actually complete the learning process, the user must at least go through all the characters in the database. Not each character box must be written in, but the user must at least scroll through the entire list by selecting the AGAIN option in the learning menu. The AGAIN option tells the algorithm that the user is done with the current learning character set and wishes to see the next set of twelve characters. If the current screen is the last set, then every character to be learned is inserted into the database. The other options in the learning menu are the BACK option, which allows the user to delete the last stroke written, the CLEAR option, which deletes all the strokes written on that character set screen, and the EXIT option, which causes the user to exit the learning session without having updated the database.

The learned characters are all inserted into the database provided that the database is not full. The database size can be determined by the user; a typical database is less than 15000 bytes. Typically, each character must be learned at least once, with most lowercase and some uppercase letters needing to be learned twice. A character takes 84 bytes to store when the algorithm is run solely on the PC, and the breakdown will be shown later in this chapter.

Note that for the learning session, the lower frames are shaded and not allowed to be written in unless the character has a descender. In Figure 4.4, only 'p' and 'q' may use the lower frame.

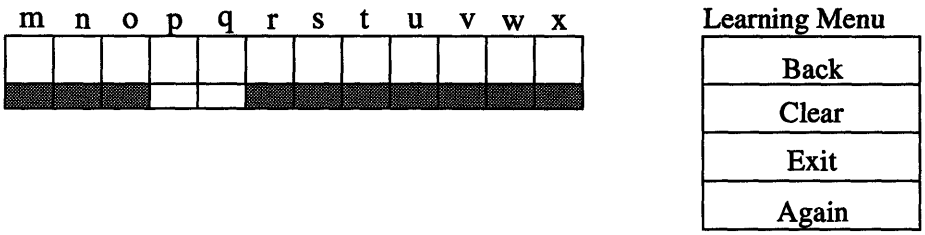


Figure 4.5 HPCR User Interface: Learning Menu

The INSERT option enables the user to enter a character that was misrecognized or not recognized by the recognizer into the symbol database. This option improves future recognition accuracy. When the user selects the INSERT option, the user is prompted to select the written character to be inserted into the database. Once the character is selected, the user then uses the QWERTY keyboard to select the proper recognition identifier of the written character, as the example in Figure 4.6 illustrates. If the identifier that is chosen is not already in the symbol database, the user is requested to verify that this new symbol should be entered into the database. If the character and its new recognized symbol are already in the symbol database, then nothing is inserted; otherwise, the symbol and its written stroke information are inserted into the database. For example, in Figure 4.6, the character written was an 'h'. The recognition algorithm, however, recognized it incorrectly as an 'L'. This misrecognition can be corrected by inserting the written character along with the correct identifier into the database. Thus, the user selects the identifier and, if the database is not full and the character symbol is valid, then the character is inserted into the database. At the same time, the misrecognized identifier, in this example, 'L', is marked against, meaning that it caused a misrecognition. If a character in the database has a certain number of markings above some user-defined threshold, then the character is eliminated from the database.

The ZOOM option enlarges a single character and allows the user to view the character data after being processed by the algorithm. The character data is the NxN character matrix that is constructed during the preprocessing stage after scaling, interpolation, and thinning. The PRINT option allows the user to print the screen to a file to be saved for future use, and the DONE option indicates to the recognizer that the recognition is finished and exits the program.

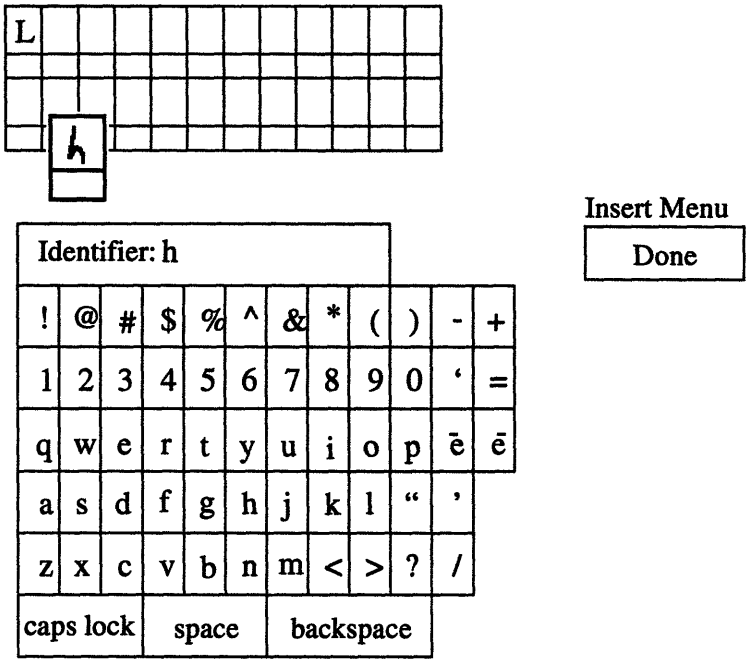


Figure 4.6 HPCR User Interface: Insert Menu

4.3 Algorithm Code

The HPCR algorithm was originally written entirely in C code. This decision was made to allow for flexibility in porting the software to different platforms and for ease of understanding. The fact that the code was written in C does have these advantages, but the primary drawback is that the code is not optimal in terms of speed. This is particularly true when porting the C code to the DSP chip. The C compiler for the DSP chip, although smart in some optimizations, still lacks the creativity and ability to fully optimize the C code for speed. The next chapter will describe the optimizations that were made to increase recognition speed.

The algorithm code can be split up into four primary blocks:

1. Preprocessing and topological engine functions
2. Recognition functions
3. Graphics functions
4. Tablet and stylus i/o interface functions

The code consists of many files, each fitting in one of the above four categories. The variable data in these files are stored in four primary structures in C. The first structure is the zone structure and contains the zone information, which includes the data for each box in the zones. This structure is filled as characters are written in a zone and is updated

throughout the recognition process. The second and third structures are the point and stroke structures, which contain the coordinates of each point and contain the stroke data such as the number of strokes. The point structure is updated as each point is written on the tablet. The bus card sends the information to the PC which then proceeds to send the necessary coordinate data to the EVM. The stroke structure is similarly filled during the writing process. The last structure is the character structure which contains the character data such as the topological engine information; a character structure is created for each learned character and stored in the symbol database. Thus, the database consists of one character structure for each character, each requiring 84 bytes of storage when the algorithm is run solely on the PC. It occupies a slightly higher number of bytes when the algorithm is ported to the EVM, and this will be discussed in more detail later. The character structure is composed of the following: 20 bytes for the features, 27 bytes for the dynamic information, 29 bytes for the microfeature information, 1 byte for the microfeature count, 1 for the character height, 1 for the character width, 1 for the misrecognized marking counter, 1 for the zone identifier, 1 for the number of strokes, 1 for the frame count, and 1 for the recognized identifier. These four structures described above are the primary data structures that need to be passed between the PC and EVM.

The topological engine data, which is stored in the character structure and consists of the features, microfeatures, and dynamic information, all are compacted by using bit fields. The bit fields are used in order to save memory; because the algorithm is company proprietary, the following examples will use numbers that are not accurate but instead are used to convey the idea of why bit fields are so useful. For instance, there are M feature maps, each producing a feature score. Suppose, for example, each feature score were each stored in one byte; then, the features would take M bytes for storage. But because the feature scores are stored in the minimum number of bits possible and because bit fields are used, then the actual number of bytes required to store the features is reduced. For this example, suppose that each of the M feature scores can be stored in 4 bits. Then, the number of bytes required to store the features is no longer M but rather M/2. Similar memory conservation occurs with the dynamic information and with the microfeatures. There are, for example, 45 microfeatures and each takes, for example, 5 bits to store. So, these are stored in bit fields, and since the PC allows bits to be stored across byte boundaries, only 29 bytes are required for storage. This bit field storage becomes more important on the 16 bit digital signal processors, as is discussed later.

4.4 Porting the C code to the EVM

The first issue that has to be resolved when porting the software to the EVM is which functions the EVM should perform and which the PC should perform. The DSP is better-suited in some tasks while the PC is better for others. The DSP's strengths are handling mathematically intensive and data intensive code. The PC is better-suited for input/output functions and graphics interfacing functions. This natural division of labor provides a logical choice in deciding which tasks each would perform. The DSP will do the actual work of preprocessing the characters and performing the character recognition, while the PC will perform the more menial tasks, for which it is better-suited, of managing graphics

and providing the communication buffer between the tablet and the PC. Thus, the PC's role is to shield the DSP from having to interface with the user and allow the DSP to focus solely on the processing and recognition of characters. Figure illustrates the division of labor chosen for this implementation between the PC and DSP.

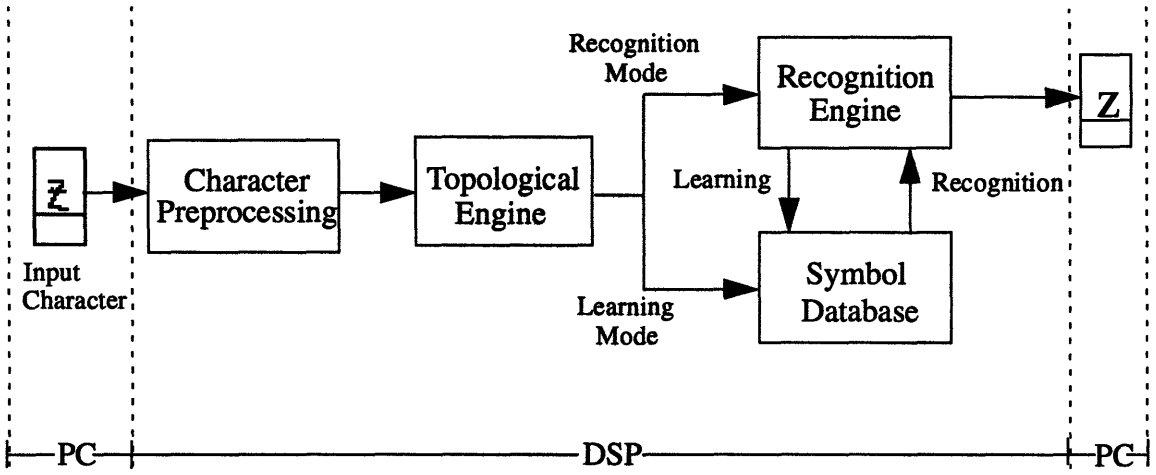


Figure 4.7 Division of labor on PC and EVM

So, after determining both the DSP's and the PC's tasks, the next step is twofold. First, it has to be decided where to place the information on the DSP and EVM, and then it needs to be determined how to link the EVM with the PC. The EVM/PC communications need to be set up so that the appropriate data transfers will occur and the PC and EVM will follow the correct handshaking protocol, as described in detail in Chapter 2. The next subsections discuss the various issues surrounding the porting of the algorithm to the EVM.

4.4.1 Memory on EVM

The EVM has only 64K words of off-chip external SRAM and on the TMS320C50, there is 9K words of single access RAM and 1K words of dual access RAM. The program code consisting of the recognition functions and the preprocessing functions occupy about 8150K words of memory on the EVM, while the variable data such as the four primary data structures mentioned above and the symbol database occupy about 44250K of memory. The stack, which stores the space for the local variable declarations within functions, requires about 3300K words. In the initial unoptimized implementation of the algorithm on the EVM, only the external memory was used and zero-wait state was selected in order to minimize the amount of cycles spent in data accesses. The program code, the variable data, and the stack were placed consecutively in the external memory on the EVM as shown in Figure 4.8.

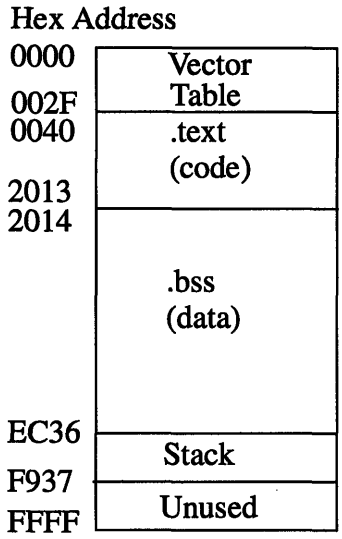


Figure 4.8 Memory Map on EVM

4.4.2 Data Constraints

There are several differences of the data between when the algorithm is run solely on the PC and when the algorithm is ported to the EVM. On the PC, the size of the data types is different from that on the DSP. On the PC, short integers and chars are 16 bits and long integers are 32 bits. The DSP does not allow a smaller division of size that 16 bits. So, short integers are still 16 bits and long integers are still 32 bits, but chars now are 16 bits as well. This issue must be taken into consideration when passing data between the PC and the DSP. The 8 bit chars on the PC side will be sent along the 16 bit data bus to the DSP, which will receive the char as 16 bits, or one word. When returning the char to the PC, the DSP sends the 16 bits back to the PC, but the PC must receive it as a char, or 8 bits. If the PC attempts to receive the char from the DSP as 16 bits, then problems could arise with overlapping data. Suppose, for example, the PC has an array of structures of characters, and it wants to fill the array from the characters sent by the DSP. If it receives each character as 16 bits from the DSP, then each time, the newly received character will overlap the others because chars are only 8 bits on the PC and the characters are being received as 16 bits from the DSP.

The problem with the 8 bits/16 bits is further illustrated in the following example. The passage of data between the PC and the DSP requires that the data transfer be done 16 bits at a time, because that is the limitation placed by the data bus. On the PC, suppose there is a structure consisting of two chars. In order to minimize the number of data transfers, the ideal solution would be to create a union on the PC of the two 8 bit chars with an integer, and then send that one integer to the EVM. Unfortunately, the same union cannot be created on the DSP. This is because on the DSP, all chars are 16 bits, so the union would actually be a union of two 16 bit chars with one 16 bit integer. Thus, by sending an integer from the PC to the DSP, not enough information would be sent and the data transfer would

fail.

Another issue that needs to be addressed is the ability to access bit fields on the DSP. The PC allows a very natural way of accessing bits, and bit fields may be stored across byte boundaries. Thus, for the microfeature data, for example, there are 45 microfeatures, each consisting of 5 bits. Again, these numbers are not accurate and are used merely for illustrative purposes. The 45 5 bit values are stored such that all of the microfeatures may be stored in 29 bytes when the algorithm is run solely on the PC. This calculation is shown below.

45 microfeatures
x 5 bit
225 bits

=> (225 bits) / (8 bits per byte) => 29 bytes

On the DSP, bit fields may not be stored across the 16 bit boundaries. Thus, rather than compacting all 45 scores into consecutive memory locations, three of the 5 bit scores would be compacted within one word, with the most significant bit being empty and unused. The next 3 of the 5 bit scores would then be compacted in the next word, and this would continue until all 45 scores were stored in 15 words. Thus, 15 words, or 30 bytes, are required for storage, which is one more byte than what is required when the entire algorithm is on the PC.

On the PC, there are 27 bytes required for storage of the dynamic information. Recall that Q points are examined, and three sets of dynamic information are extracted from these Q points. The data are stored in sets of 8 bits, and there are exactly 8*27, or 216, bits required for dynamic information storage. On the DSP, because there is no smaller division than one word, 14 words or 28 bytes, which is actually 224 bits, are required for storage. Thus, on the DSP, each character structure, which took 84 bytes of storage on the PC, now takes 86 bytes of storage after porting the algorithm to the EVM. Identical data structures are created on the PC and the EVM. This facilitates the data transfers and the communication process. Thus, after porting the algorithm to the EVM, the PC now requires 30 bytes to store the microfeatures and 28 bytes to store the dynamic information, and it requires 86 bytes in all to store the entire character structure.

4.4.3 EVM Communication

The data transfers between the PC and the EVM must be coordinated so that each knows exactly what the other will be doing. This is particularly crucial because the EVM cannot send commands to the PC; as discussed in Chapter 2, only the PC may send commands. Thus, the data that is sent must be known specifically by both sides to avoid any incomplete data transfers. The first step was to thus determine the data that the PC would need and the EVM would need to perform their respective tasks. The data could be transferred only 16 bits at a time, which can be a fairly time consuming process. Thus, the goal was to minimize the amount of data that needed to be transferred.

The communication between the PC and EVM has to be exact. Each has to know exactly what the other is going to do next, otherwise there will be a break in the handshaking sequence and the communication will be halted or stalled somewhere. The PC sends commands that will tell the EVM what to do next and what data to receive. The command also signals to the EVM which data the PC expects back, and thus there is a perfect data transfer sequence.

Because of the difficulty in the 16 bit issue, much care has to be taken when sending the bit-fields. The solution is to create unions of these bit fields with 16 bit integers. For example, the microfeatures are stored in an array of structures. The array contains 15 elements, and each structure contains three 5 bit fields. In order to send this data to the EVM, each structure is linked in a union with an integer. The array now is an array of 15 unions of a structure and an integer. In this manner, the integer can just be sent back and forth between the PC and the EVM. Similarly, when sending a long integer, the long integer cannot be sent entirely at one time because of the 16 bit data bus. A union can be created between the long integer and 2 short integers. The 2 short integers can then be sent to the EVM from the PC and received by the EVM, which will have the identical variable declaration. This way, the long integer would be transferred properly.

Note that no union need be created for chars. Recall that they are 8 bits on the PC but 16 bits on the DSP. So, the char can be sent entirely at one time from the PC to the DSP. When returned, the only precaution that need be taken is to receive the data as 8 bits to avoid rewriting data.

Once, the data is sent correctly to the EVM, the DSP can go about doing its preprocessing and performing the recognition. Then, the EVM expects a command from the PC, and the EVM would send the appropriate data. An example of the sequence of commands and actions when recognizing characters in the single character writing zone is shown in the following table. The table shows the typical chronology of events on the host and DSP as a character is written and then asked to be recognized. The sequence of events, from top to bottom, is from the beginning of the recognition process to the end. Horizontal division of the table indicates a causal and temporal relationship between the various stages.

Table 4.1 Command Hierarchy Table

HOST ACTIONS	DSP ACTIONS
(H1) Host sends command to DSP to start process of sending preprocessing info to DSP	
	(D1) DSP receives all the data and then performs preprocessing. After, it waits for host to send the return data command
(H2) Host sends command to DSP to get DSP to return necessary info to host	
	(D2) DSP returns preprocessing data. The above steps are repeated for each box with written characters in it in the single character writing zone. Then, waits for the begin recognition command
(H3) Host sends command to begin process of sending appropriate data for recognition to DSP	
	(D3) DSP receives data and performs recognition. After, it waits for the host to send the return data command
(H4) Host sends command to DSP to get DSP to return necessary info to host	
	(D4) DSP returns appropriate recognition data. (D3, D4) are repeated for each box and then the DSP waits for next command

Thus, this is one command sequence that occurs during the recognition system. This table illustrates the typical sequence of events: the PC sends a command to the EVM and the EVM performs the task. The complete list of commands is shown in the following table along with their function. These commands are the only ones uses by the PC to communicate with the EVM.

Table 4.2 Commands to EVM

Command	Function
NOCOMMAND	A dummy command used when the data transfers between the PC and EVM are complete
GET_PREPROCESS_DATA	Tells the EVM that PC is about to send data essential to starting preprocessing
GET_RECOGNITION DATA	Tells the EVM that PC is about to send data essential to starting recognition
GET_DATABASE	Tells the EVM that PC is about to send symbol database
RETURN_PREPROCESS_DATA	Tells EVM to return preprocessing data
RETURN_RECOGNITION_DTAA	Tells EVM to return recognition data
DONE	Tells EVM that the program is finished and the EVM can quit

4.4.4 Profiling

After porting the code to the DSP, the next step is to calculate the number of cycles the DSP takes to perform the preprocessing and the recognition. This step is known as profiling. What is used as input was a file that contains the stroke information that can be used as the input to the preprocessing. This off-line stroke information allows for the preprocessing and recognition to be performed on the same input each time and thus provides a reliable method of comparison. The next chapter on optimization will use the same input and it is then possible to compare quantitatively the improvements made. The stroke information is sent to the preprocessing and topological engine unit, and afterwards, the recognition is performed. The results from this profiling are shown below. The profiler allows the user to determine how many instruction cycles a specific line, range of lines, function, or program takes. The output from the profiler is the number of cycles taken, and for this algorithm, was performed on the main preprocessing and main recognition functions on the EVM. The number of cycles shown in Table 4.3 is for one character being processed and recognized. The TMS320C50, as mentioned in Chapter 2, has a 25ns instruction cycle time, so the time spent in the preprocessing stage is about 4.15 ms and in the recognition stage it is 10.46 ms, for a total recognition time per character of 14.6 ms. This time is a fairly significant reduction to that obtained on the PC, using a 486 33MHz computer, which achieves a recognition time of 20 ms per character. Furthermore, the accuracy of the algorithm is very high, and tests performed on this algorithm yield an accuracy of greater than 99%.

Table 4.3 Profile Results

Function	Number of Cycles
Preprocessing	165848
Scaling/Interpolation	82029
Thinning	15695
Feature Extraction	22975
Microfeature Detection	11979
Dynamic Information	28821
Recognition	418482
Compare # strokes	3324
Compare width/height	2636
Compare features	150216
Compare microfeatures	163231
Compare dynamic info	92039

4.5 Portability

The goal of the implementation of the algorithm onto the EVM is to verify the functionality of the of the recognition system when being run primarily on a DSP chip. The final platform will not, however, be the EVM. Instead, it will be some other platform that uses the Texas Instruments TMS320C50. Thus, the code on the EVM has to be made modular in order to facilitate future porting.

The code is thus split up into different modules, each performing a separate and distinct function from the others. The modules are:

1. Core routines
2. Input/Output routine
3. Initialization
4. Test vectors
5. Control Routine

The core routine does the bulk of the actual work and includes all the functions that

implement the core algorithm. These functions are independent of any hardware constraint and the only target specific information they possess is that the target is Texas Instruments' TMS320C50. For this character recognition system, the preprocessing and the recognition functions were the core routines. The input/output routine is the routine that provides communications between the EVM and the PC. The initialization routine includes any functions that perform hardware or software initialization. As described in Chapter 2, the EVM communications require initialization before beginning the communication process. The control routines consists of all the functions that control the program flow or control the command handler. These functions may contain some hardware specific information but they mainly just control the program flow. Finally, the test routines are used to ensure that any porting of the algorithm has not affected the integrity of the algorithm or the program.

4.6 Conclusion

The hand printed character algorithm is now run in real-time on the DSP chip. The DSP performs the processing and recognition functions while the PC performs the interfacing and input/output functions. The algorithm, although functional, is not optimized for speed; the next chapter will discuss the various optimizations made to increase recognition speed. The recognition times achieved by both the PC and the DSP are both very fast and allow for real-time recognition. A simple test to prove that even the 20 ms achieved by only the PC is fast enough to allow for real-time recognition is to write segmented characters as fast as possible. A simple test of several people yielded an average writing speed of about two or three characters per second, with the fastest being about six or seven characters per second when writing a '1'. Thus, given this speed for entering text, it is apparent that the PC alone can support a real-time system. However, the purpose of optimization, which is discussed in the next chapter, is to not only make the recognition as fast as possible for real-time recognition, but also to ensure that any off-line recognition occurs as fast as possible when using this system. Off-line recognition might be required in this system if the user wishes to completely write the text first and then submit that text for recognition; in this case, it is optimal for the algorithm to run as fast as possible.

Chapter 5

Optimizations

5.1 Introduction

The purpose of using the digital signal processing chip as the processor for the hand printed character recognition algorithm is to allow fast, real-time recognition. The implementation is such that the DSP chip is responsible for the mathematically and data intensive preprocessing and recognition functions, while the PC handles the input/output and graphics tasks. In order to implement this algorithm using the DSP chip, the EVM was used as the platform to test the recognition; the EVM contains external memory along with the TMS320C50 DSP chip. The code for the algorithm can then be run on the DSP chip by translating the source program that contains the processing functions into code that the chip can execute. The original algorithm is written in C, so the C compiler can be used to translate the code into assembly language source code. An assembler can then take these assembly language source files and translate them into machine language object files, and the final step is to link these object files into a single executable object module.

The C compiler translates the C code into assembly with the ability to perform somewhat sophisticated optimizations that generate efficient, compact code. These optimizations, while producing code significantly faster than the unoptimized code, still has room for improvement in terms of optimization for speed. Thus, the next step in implementing the hand printed character recognition algorithm in real-time is to optimize the code by rearranging and rewriting the assembly code generated by the compiler. The purpose is to create assembly programs that can perform the equivalent tasks as the original assembly files in fewer cycles. There are several basic, general optimizations that can be performed to save a significant number of cycles, as well as code-specific optimizations that can reduce the number of cycles.

This chapter will explore these issues surrounding optimization and attempt to illustrate several examples of optimization performed for the hand printed character recognition algorithm. Section 5.2 will first give a brief overview of the assembly language instruction set for the TMS320C50 and provide a description of relevant instructions that will be used in this chapter. Section 5.3 will then describe the C compiler and its behavior during execution, which will provide insight in the optimizations that are possible and necessary. Section 5.4 will then briefly mention several general optimizations that can be performed to save cycles, and Section 5.5 will describe, more in depth, several specific optimizations made in this thesis. Finally, section 5.6 will illustrate the significant reduction in cycles achieved through optimization.

5.2 Assembly Language on the TMS320C50 chip

The TMS320C50 chip supports a base set of general purpose assembly instructions as

well as arithmetic intensive instructions that are particularly suited for digital signal processing and other numeric-intensive applications. These assembly instructions can be grouped in the following sections: control instructions, I/O and data memory instructions, branch instructions, multiply instructions, parallel logic unit instructions, auxiliary register instructions, and accumulator instructions. Table 5.1 lists relevant instructions, along with a description for each, that will be used in this chapter [31].

Table 5.1 Relevant Instructions

Instruction	Description
ABS	Absolute value of accumulator
ADD	Add to accumulator
ADDB	Add accumulator buffer to accumulator
AND	Perform bitwise AND with accumulator
LACC	Load accumulator
SACB	Store accumulator in accumulator buffer
SACH	Store high part of accumulator
SACL	Store low part of accumulator
SUB	Subtract from accumulator
SUBC	Conditional subtract used for division
LAR	Load ARx
SAR	Store ARx
MAR	Modify ARx
ADRK	Add to ARx a short immediate constant
SBRK	Subtract from ARx a short immediate constant
MPY	Multiply
B	Branch
RET	Return from subroutine
CALL	Call subroutine
RPTB	Repeat block of instructions
BIT	Test bit

Most of these assembly instructions are one word long and can be performed in one cycle. A multiply instruction, for example, takes one cycle to execute, as does an addition and a subtraction. There are a few instructions, however, that take multiple instructions to execute. A branch, for example, take four cycles to execute. A branch instruction takes as an argument the location to branch to, and it then transfers the system control to that location. It is thus necessary to flush the pipeline to allow the new branch address to fill the pipeline. Because the TMS320C50 has a four-deep pipeline, there are four cycles required to execute the branch. Instructions such as CALL and RET also take four cycles, and logical instructions such as AND take 2 cycles; most other instructions, though, can be performed in a single cycle, and this characteristic enables the DSP to efficiently execute code.

5.3 The TMS320C5x Compiler

The C compiler takes C source code as input, and it translates each line of code into assembly [33]. The output is thus assembly language source files that can then be assembled and read by the DSP chip. In the translation of C to assembly, the compiler uses a stack for several purposes. First, the stack is used to allocate local variables in a function. These temporary variables can be stored in the stack during a function's execution, and they can then be discarded or written over upon the function's termination. The stack is also used to pass arguments to a function; this will be discussed further below. Finally, the stack saves the processor status, so functions can be called and then returned to the correct location.

The compiler follows a strict convention for register usage, function calling, and function termination. This convention must be understood and followed when rewriting assembly code. The compiler employs two registers that should not be modified during a function's execution, AR0 and AR1. AR0 is the frame pointer and points to the beginning of the frame; a frame is the current function's available portion of the stack and is created on the stack for each function. AR1 is the stack pointer, and it points to the top of the stack. Both of these registers are crucial in allowing the program to properly enter and return from functions. There are other auxiliary registers that also may be used by the compiler, depending on the user. Auxiliary registers AR6 and AR7 may be used to store register variables, and these registers must be properly saved and restored as described below if used. Normally, register variables are declared in C, and they should only be used if the variables will be used frequently, as the initialization of the register variables in assembly takes about four cycles. All other memory-mapped registers may be used during a function's execution without having to be saved and restored.

During a function call, there are strict rules imposed by the compiler. Arguments passed to the function must be stored on the stack before the function is called; these arguments must be pushed onto the stack in reverse order, so the rightmost argument is pushed first and the leftmost argument is pushed last. For each function call, the compiler builds a frame to store information on the stack. The current function's frame is called the local frame. The C environment uses the local frame for saving information about the caller,

passing arguments, and generating local variables. Each time a function is called, a new frame is generated to store information about that function. The compiler performs the following tasks when building a local frame:

1. Pushes the return address onto the stack
2. Pushes the contents of the old frame pointer, AR0, onto the stack and then sets the new frame pointer to the current stack pointer
3. Increments the stack pointer by the number of words needed to hold local variables
4. If the function uses register variables, saves the values of AR6 and AR7 onto the stack.

Figure 5.1 illustrates the stack after a function has been called.

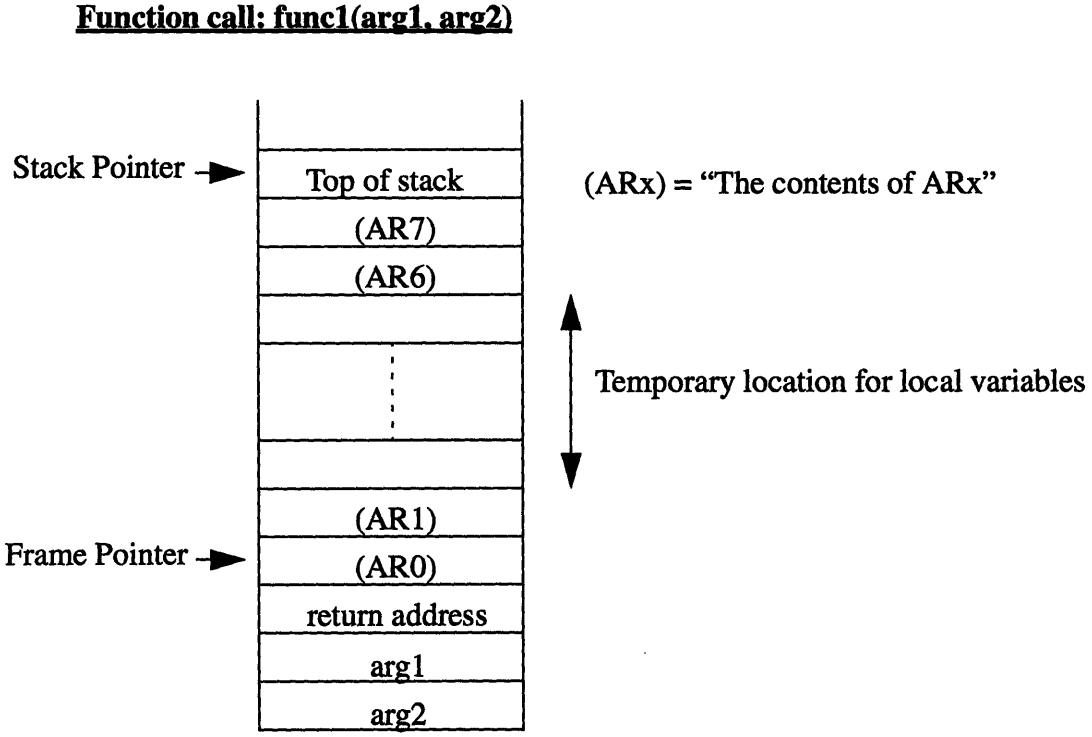


Figure 5.1 Function Calling Convention

During function termination, the above steps are reversed. The data stored on the stack during the function call can now be used to restore the values of the registers and allow proper return to the caller function. If data is to be returned by the function, it is stored in the accumulator. The arguments are not popped off by the called function but instead by the caller function, which thus allows a variable number of arguments to be passed to the called function without the called function having to know how many variables were passed.

The above conventions are critical and must be followed when rewriting the assembly code. Another aspect of the compiler that should be understood in order to begin the optimization of the character recognition code is how the compiler optimizes the code. The compiler has the capability of improving the execution speed and reducing the size of C programs by doing such things as simplifying loops, rearranging statements and expressions, and modifying data expressions. Some of these optimizations are described below.

One of the simple optimizations that the compiler performs is to use the DSP chip's ability to support zero overhead loops with the RPTB (Repeat Block) instruction. This instruction allows loops controlled by counters to be translated into assembly with no cycles being wasted with loop counters. There are other instructions that can also be utilized by the compiler to save cycles. Specifically, the compiler can use delayed branches, calls, and returns. The branch unconditionally delayed (BD), call delayed to a named function (CALLD), and simple return delayed (RETD) are the three instructions that the optimizer uses. These instructions save two cycles by allowing two instruction words to be executed after the delayed instruction enters the instruction stream. By allowing two one-word instructions or one two-word instruction to follow the delayed instruction, the compiler effectively saves two cycles.

The optimizer also saves cycles by modifying loops. The optimizer performs loop rotation by evaluating loop conditions at the bottom of the loop, thus saving a costly extra branch out of the loop. The optimizer also identifies expressions within loops that always compute to the same value. This computation is moved in front of the loop, and each occurrence of the expression in the loop is replaced by a reference to the precomputed value. A third loop modification that the optimizer performs is loop unrolling. This step saves cycles in reducing the number of times a loop is executed by repeating the code in the original loop multiple times within a single new loop cycle.

Branch optimizations are another way the optimizer saves cycles. The compiler analyzes the branching behavior of a program and rearranges the basic blocks to remove branches or redundant conditions. Unreachable code is deleted, branches to branches are bypassed, and conditional branches over unconditional branches are simplified to a single conditional branch. When the value of a condition can be determined at compile time, a conditional branch can be deleted.

Finally, the optimizer performs some data optimizations. When the same value is produced by two or more expressions, the compiler computes the value once, saves it, and then reuses it. The compiler also simplifies expressions into equivalent forms requiring fewer cycles. For example, the expression (a+b) - (c+d) takes 6 cycles to evaluate. Each instruction necessary is executed in one cycle, but it takes the following six instructions to execute:

```
1. LACC c     ;Loads the accumulator with the value in c
2. ADD d      ;Adds the value in d to the accumulator
```

```
3. SACL AR4   ;Stores the result of c+d in AR4
4. LACC a     ;Loads the accumulator with the value in a
5. ADD b      ;Adds the value in b to the accumulator
6. SUB AR4    ;Subtracts the value of (c+d) from (a+b)
```

With the optimizer, the compiler optimizes this to ((a+b)-c)-d, which takes only 4 one-cycle instructions:

```
1. LACC a     ;Loads the accumulator with the value in a
2. ADD b      ;Adds the value in b
3. SUB c      ;Subtracts the value in c
4. SUB d      ;Subtracts the value in d
```

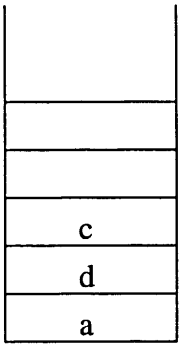
These optimizations, while providing a large improvement over the unoptimized code in terms of speed, still does not fully optimize code and leaves room for improvement. The next section will describe several basic, general optimizations that can be made to save even more cycles.

5.4 Basic Optimizations

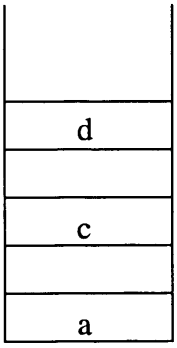
There are several basic optimizations that can be made that the optimizer cannot perform. This section will describe various general optimizations that can be made to most applications, while the following section will detail several specific optimizations that were also employed in optimizing the hand printed character recognition algorithm. First of all, many cycles can be saved by using all available registers as storage locations for variables and constants rather than using the stack. The stack and indirect addressing allow for efficient accessing of data when the data is in array format; i.e., if the data is to be accessed consecutively on the stack, then the stack is the ideal storage location for the data. If, however, the data or variables on the stack cannot be arranged so that they are accessed consecutively, then cycles are wasted by incrementing the pointer to the correct location on the stack. Indirect addressing provides the capability of incrementing the pointer to the stack by one without a loss in cycles. But incrementing by more than one requires extra cycles, and these cycles can be eliminated when using registers and avoiding the stack. Figure 5.2 illustrates the problem.

The optimizer is only knowledgeable of using the auxiliary registers to store data. There are, however, many more memory mapped registers that may be used, depending on the application. As mentioned in Chapter 2, there are twenty-eight memory mapped registers in all, and if these registers are not used for their designed purpose, they can be used to store temporary variables and they need not be saved and restored. For example, if circular buffer addressing is not performed in the application, then the registers CBSR1, CBSR2, CBER1, and CBER2 can all be used as temporary storage registers for variables and data.

C Statement: a = c + d




```
LACC *-   ;Load accumulator with c
ADD  *-   ;Add d
SACL *    ;Store result in a
```




```
LACC *   ;Load accumulator with c
ADRK 2   ;Add 2 to pointer to point to d
ADD  *   ;Add d
SBRK 4   ;Subtract 4 from pointer to point to a
SACL *   ;Store result in a
```

Figure 5.2 (a) Data ordered consecutively in stack efficiently uses indirect addressing
(b) Data ordered non consecutively takes more cycles using stack

Another basic optimization that can be performed is to eliminate redundant variable accesses. It is fairly costly to access elements in a structure because of the cycles that must be spent getting the base address of the structure and then adding the offset to this base address to get the appropriate element. Thus, it would save cycles to eliminate any redundant accessing of the structure's base address and the adding of the offset for various elements of the structure by simply getting the addresses of all the elements at once and storing the addresses in registers or in the stack. Suppose, for example, that one of the arguments to a function is a structure with two integers as its elements. If the two elements are accessed at different times in the function, then there are redundant accesses performed in getting the structure's base address and adding the necessary offset. If both accesses are done at the same time so that the address of each element in the structure is obtained and stored, even for this simplified case, a significant reduction in cycles occurs. Figure 5.3 provides an example illustrating this scenario.

```
typedef struct {
  int element1;
  int element2;
} example_struct;


void function(example_struct var1)
{
    int local1 = var1.element1;
                 :
                 :
                 :
    for ( ...)
    {
       int local2 = var1.element2;
                    :
                    :
                    :
    }
}
```

Figure 5.3 C function causing redundant variable access of local1 and local2 in assembly

From the figure, the accessing of variables local1 and local2 in assembly are performed in two separate locations. For each access, the following is performed:

2 instructions to move the pointer of the stack to point to the address of var1
1 instruction to load the address into the accumulator
1 instruction to add the offset to get the address of the appropriate element
1 instruction to store the address

---

5 cycles/variable access = 10 total cycles for local1 and local2

If, instead, the variables are accessed at the same time, then the code can be optimized so as to save redundant accesses:

2 instructions to move the pointer of the stack to point to the address of var1
1 instruction to load the address into the accumulator
1 instruction to add the offset of the correct element
1 instruction to store the address of var1.element1
1 instruction to store the address of var1.element2

---

6 total cycles for local1 and local2

Thus, 4 cycles are saved by this simple optimization. This type of optimization is particularly useful when the structure is very large and the variable accesses are scattered throughout a function. The savings in cycles then becomes more significant.

Another optimization involves using some instructions that are not utilized by the optimizer at all. For example, the RPT (Repeat Next Instruction) instruction is not ever used by the compiler; instead, only the RPTB instruction is used. The RPTB instruction, though, requires at least 3 instructions to follow it. If however, only one instruction is to repeated, then this is a very inefficient execution:

```
RPTB
  SACL *+
  NOP
  NOP
```

There is also an overhead in setting up the RPTB statement. The number of times the block is to be repeated must be stored in the BRCR register, and this storage costs two cycles.

The alternative and more efficient method is to use the RPT statement:

```
RPT #times
 SACL *+
```

Finally, there are branch conditionally delayed (BCNDD) instructions. These are similar to the branch unconditionally delayed instruction (BD), except these conditional branch instructions branch only when certain conditions are satisfied, such as if the accumulator is greater than zero or if the accumulator is less than zero. These then branch if the condition is satisfied, otherwise they do not. These instructions have the ability to execute the 2 instruction words following the branch conditionally delayed instruction, just as the branch unconditionally delayed instruction behaves.

The above are examples of general optimizations that can be made on most applications; the next section details three specific optimizations performed for the hand printed character recognition algorithm.

5.5 Specific Examples of Optimizations

5.5.1 Bit Field Manipulation

The digital signal processing chip is not particularly well-suited for bit field manipulations. There are, however, some bit field manipulations that it must perform in order to process the feature extraction, microfeature detection, and dynamic information. Recall that these information are compacted in bit fields in order to conserve the memory

required to recognize a character. Thus, a 16-bit word can contain 8 2-bit feature map scores, the microfeature scores for one row, or two sets of dynamic information.

The bit fields are created in C with a structure that contains the bit field names along with the number of bits required per field. The compiler for the DSP chip is not able to efficiently translate this to assembly. For example, when performing the recognition, it is necessary to compare the feature scores of the input character with the feature scores of each character in the symbol database. This task is accomplished by comparing, one at a time, a 2-bit feature score of a character from the database with a 2-bit feature score of the input character. The feature scores are stored in ten 16-bit words, and a word from each character is compared at one time. In order to access a single feature score within a word, the C compiler creates the following assembly code:

```
LACC *, 13, AR1 ;Load ACC with a word containing 8 feature scores
AND  #0003h, 15 ;Perform bitwise AND to get desired 2 bits of word
SACH *, 1, AR5  ;Store feature score in stack
LACC *, 13, AR1 ;Load ACC with a word containing 8 feature scores
AND  #0003h, 15 ;Perform bitwise AND to get desired 2 bits of word
SUB  *, 15      ;Subtract the feature score stored in stack
SACH *, 1       ;Store high part of accumulator (result) in stack
LACC *, AR6     ;Loads the result back in accumulator
ABS             ;Takes absolute value of subtraction
ADDB            ;Adds the previous result in ACCB to ACC
SACB            ;Stores the new result in the ACCB
```

These steps are illustrated in Figure 5.4.

The steps take, in all, 13 cycles, because each step takes one cycle except for the bitwise AND operations which take 2 cycles. This method is repeated for all 80 feature scores, and this is thus a very time consuming process. Though there are limitations to the reduction in complexity to perform the bitwise operations, some cycles can be saved. The above steps are repeated for the remaining 7 feature scores in the word, with the only change being the shift required for the AND operation. Thus, these 13 cycles are repeated for all 80 feature scores, resulting in 1040 cycles. Furthermore, these 1040 cycles are repeated for each character in the symbol database. A typical database has over 100 characters, and for this implementation, about 125 characters. So, this bit field manipulation would take 130000 cycles.

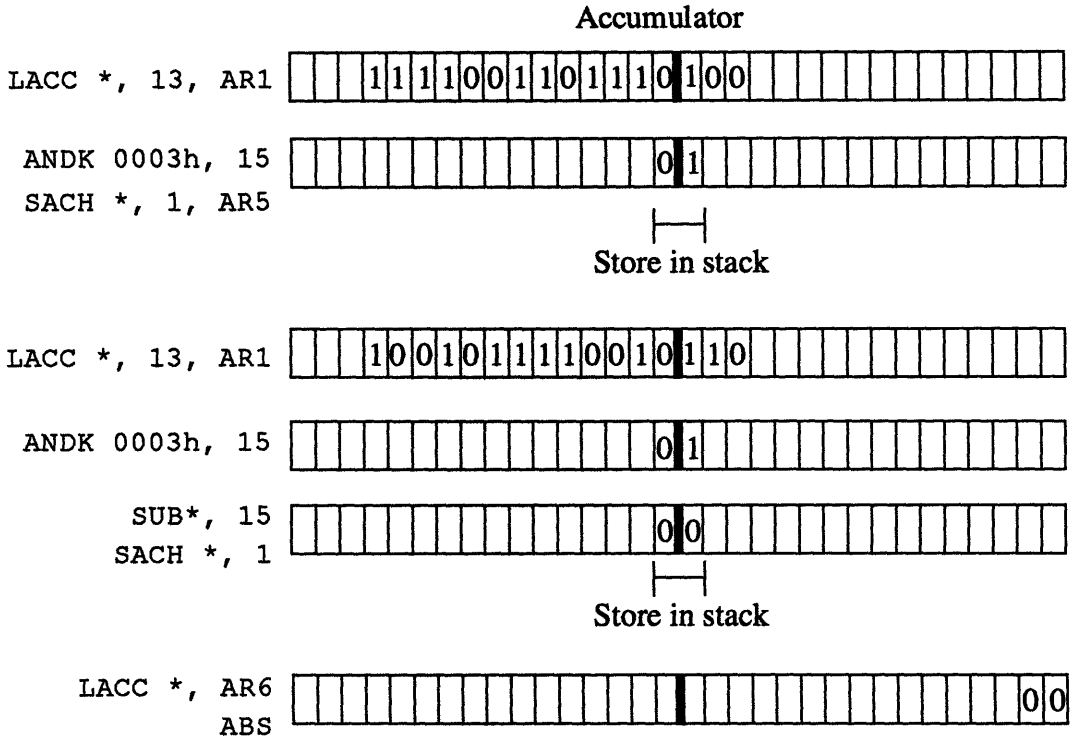


Figure 5.4 Optimized compiler generated bit field manipulations

A simple rewrite of this short assembly code can save a significant number of cycles. This example is used in this chapter because of its simplicity to describe. Both the rewritten functions of the microfeature detection recognition scoring and the dynamic information recognition scoring required a bit more complicated optimizations and resulted in more cycles saved. This example suffices, however, in describing the general principle used in saving cycles for bit field manipulations. Thus, the following rewrite was made to the feature extraction recognition scoring function.

```
LACC AR3, 2    ;Load ACC with a word containing 8 feature scores
SACH DBMR      ;Store single feature score in DBMR
SACL AR3       ;Store rest of feature scores in AR3
LACC INDX, 2   ;Load ACC with a word containing 8 feature scores
SACH ARCR      ;Store single feature score in ARCR
SACL INDX      ;Store rest of feature scores in INDX
LACC DBMR      ;Load one feature score into accumulator
SUB  ARCR      ;Subtract other feature score
ABS            ;Takes absolute value of subtraction
ADDB           ;Adds the previous result in ACCB to ACC
SACB           ;Stores the new result in the ACCB
```

This is illustrated in Figure 5.5. This takes the same number of steps, but this time each step is a single cycle. Thus, each feature score takes 11 cycles, and all 80 feature scores take 880 cycles. For all 125 characters in a typical database, this would thus take 110000 cycles, a savings of 20000 cycles. And this reduction was made by a very simple change in the way of handling bit field manipulations. What was avoided was the use of the bitwise AND operation and what was utilized were more registers and less use of the stack. The use of more registers, as mentioned earlier, is one of the benefits of writing functions in assembly directly instead of relying on the compiler to translate the C code to assembly. The C compiler is not able to utilize all of the registers that a user can use. Again, more complex changes were made to the other two recognition functions, but this example illustrates a simple rewrite that can be made to perform the same function in fewer cycles.

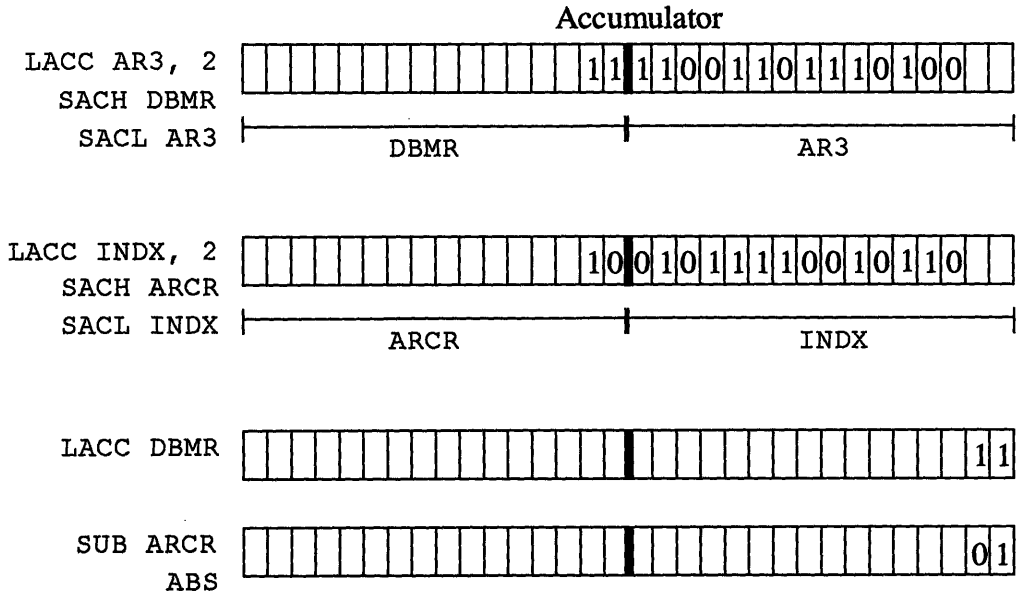


Figure 5.5 Rewritten bit field manipulation

5.5.2 Division Routine

Division is a very costly subroutine on the TMS320C50 DSP chip. There is no direct support from the chip for integer division, so all division and modulus operations are performed through calls to runtime-support routines. These calls to the division and modulus routines are costly, for many checks have to be made to ensure that the divisor is not zero and to determine the sign of the quotient. Furthermore, a typical division routine has many subroutines to call and branches to take, so the end result is a very time consuming routine. In the example in Figure 5.6, both the division routine and modulus routine must be called separately.

```
typedef struct {
  int element1;
  int element2;
} example_struct;

example_struct var1 = {3, 46};

void func(example_struct var1)
{
  int x,y;

  x = var1.element2 / var1.element1;
  y = var1.element2 % var1.element1;
}
```

Figure 5.6 Example C code illustrating inefficient separate division and modulus routines

The division operation takes about 140 cycles in assembly, and the modulus operation takes about 130 cycles to execute. These seemingly simple operations are thus very costly. Furthermore, many cycles are wasted as separate division and modulus routines must be called despite the fact that both the modulus and division routines are very similar and they both calculate the quotient and remainder during their execution. With the additional checks that must be made on the data, these routines are thus very time-consuming kernels.

Thus, because of the inefficiencies presented above, the next step was to rewrite a subroutine that could take advantage of the knowledge of the type of data and that could perform both the modulus and division operations at one time. This rewrite, like the runtime-support function, uses the assembly instruction, SUBC (Conditional Subtract), to perform the actual division. The conditional subtraction instruction is repeated sixteen times for 16-bit division, and the quotient is located in the lower 16 bits in the accumulator. The division and modulus routine checks for the sign of the divisor and dividend, but it does not check if the divisor is zero, as this never occurs in the hand printed character recognition code. It does check if the dividend is less than the divisor, in which case, because this is integer division, it returns zero as the quotient and returns the dividend as the remainder. The entire routine is shown below:

```
DIVMOD:
  ADRK  11          ;Get temporary location on stack
  SACH  *-          ;Save the sign of dividend
  ABS               ;Make dividend positive
  SACL  *, AR1      ;Store dividend
  SUB   *, AR2      ;Store divisor
  BCND  RETQUO, LT  ;Branch if dividend < divisor
```

```
  LACL   *+, AR1      ;Load accumulator with dividend
  RPT    #15          ;16 cycle division. Low accumulator=quotient
   SUBC               ;High accumulator=remainder
  MAR    *, AR2       ;Change ARP to AR2
  BIT    *, 0         ;Test the sign that was stored on stack
  RETCD NTC           ;Return if positive (do next 2 instructions)
   SACL   *-          ;Store quotient in stack
   SACH   *+          ;Store remainder in stack
  LACC   #0           ;Load 0 into accumulator
  SUB    *            ;Subtract quotient
  SACL   *-           ;Store quotient (negative)
  LACC   #0           ;Load 0 into accumulator
  RETD                ;Delayed return (do next 2 instructions)
   SUB    *           ;Subtract remainder
   SACL   *+          ;Store remainder (negative)
RETQUO:               ;Come here if dividend < divisor
  MAR    *+           ;AR2 now points to stack where sign is
  BIT    *, 0         ;Test the sign
  RETCD NTC           ;Return if positive (do next 2 instructions)
   LACC   #0          ;Load 0 into accumulator
   SACL   *           ;Store quotient in stack
  MAR    *-           ;AR2 now points to stack where dividend is
  RETD                ;Delayed return (do next 2 instructions)
   SUB    *           ;Subtract remainder
   SACL   *+          ;Store remainder (negative)
```

The quotient and remainder are both stored in the stack. From the example in Figure 5.6, the rewritten division and modulus routine takes, in all, under 40 cycles to execute. This savings in cycles is very significant, and this optimization creates a very efficient implementation of a routine that logically combines the division and modulus routines.

5.5.3 Optimizing the Memory Map

Finally, the memory map was optimized to save cycles. As mentioned in Chapter 4, the entire algorithm was initially placed in the external SRAM of the EVM. While functional, this implementation is fairly inefficient in terms of cycles spent on data and code accesses. This inefficiency arises from conflicts between instruction and operand fetches in the pipeline. When both the code and data are in external memory, there will be a conflict between an operand fetch followed by an instruction fetch. This is due to external RAM being single-access, so only one read or one write may occur in single cycle. As a result, when an operand is being fetched, an instruction cannot simultaneously be read.

This conflict causes the instruction to take one extra cycle. Furthermore, all external writes take at least two machine cycles; when an external write is followed or preceded by

an external read cycle, then the external write requires three cycles. The above reasons lead to the motivation for using on-chip memory.

The usage of on-chip memory allows either code or data to be fetched from external memory while the other is fetched from on-chip memory. This reduces data conflicts and saves cycles. This on-chip memory is divided into four 2K word blocks of single-access RAM and one 1K word block of dual-access RAM; however, one single access block allows only one access per cycle. In other words, the processor can read or write on one single access RAM block while accessing another single access RAM block at the same time. The dual access RAM is even more versatile, as it allowing both a read and a write to occur in the same cycle.

Therefore, in order to optimize the memory map to minimize the number of cycles, the goal is to place the code and data in different memory locations to reduce the number of conflicts. Thus, the following placement of the code and data is performed. The code, consisting of the primary optimized assembly language recognition and preprocessing functions is placed in the on-chip single-access RAM. This code is about 6500 words. The rest of the less utilized code is placed in external memory. The data, about 39300 words, is placed in external memory, and the stack is reduced to 1K words and placed in the dual-access RAM. This memory mapped optimization significantly reduces the number of data conflicts and allows much more efficient data and code accesses, resulting in fewer cycles wasted.

5.6 Results

From all of the optimizations mentioned, the result is a significant reduction in the number of cycles spent recognizing characters. As Table 5.2 indicates, the optimized code after rewriting and rearranging assembly code recognizes a character more than twice as fast as the code that is generated by the optimized compiler. From the profile results, the optimized code, run on the TMS320C50 DSP with a 25 ns instruction cycle, performs the preprocessing in 1.04 ms and the recognition in 4.69 ms, for a total recognition time per character of 5.73 ms. This time is a significant reduction from that in Chapter 4 of 14.6 ms, and it is even more impressive when compared to the recognition time on a 486 33MHz PC of 20 ms.

Thus, the optimization step proved to be very successful and resulted in a much faster implementation of the hand printed character recognition algorithm. As mentioned before, even the 20 ms recognition speed is fast enough to support real-time recognition. The purpose of the optimization phase was not just to ensure that real-time recognition could be performed, but it also was to enable any off-line recognition to be performed as fast as possible. With the significant reduction in recognition speed, this algorithm is now ideal for both on-line recognition and off-line recognition.

Table 5.2 Profile Results

Function	Before Optimization (code from C compiler)	After Optimization (rewritten assembly)
Preprocessing	165848	41749
Scaling/Interpolation	82029	14784
Thinning	15695	3738
Feature Extraction	22975	9884
Microfeature Detection	11979	3701
Dynamic Information	28821	6171
Recognition	418482	187786
Compare # strokes	3324	2969
Compare width/height	2636	2355
Compare features	150216	66148
Compare microfeatures	163231	75983
Compare dynamic info	92039	34046

Chapter 6

Unboxed Recognition

6.1 Overview

The recognition algorithm described thus far provides a constrained environment for character recognition; in this boxed environment, the user segments the characters by writing each character inside an individual box. This user segmentation facilitates recognition by controlling the spacing of each character and by allowing it to be easily determined which stroke belongs to which character. As mentioned in Chapter 1, the segmentation of unboxed characters is not a trivial process and is prone to erroneous separation of characters. Another advantage of boxed recognition is that it provides the constraint of limiting the height and width of the written characters; this additional constraint helps to standardize the input characters before beginning the preprocessing and recognition stages. Furthermore, with the information provided by the separation of the boxes into two frames, which allows easy division of characters with and without descenders, the boxed recognition algorithm in this thesis does achieve a very high recognition accuracy rate.

While boxed recognition can be useful and is adequate for many applications, improvements can be made to this recognition system by providing the user with additional freedom in writing the characters. However, from this added freedom for the user, a problem is introduced due to the segmentation necessary to separate the letters. This segmentation must be performed carefully so as to allocate the strokes to the correct character. One type of algorithm for unboxed recognition that will be discussed in this chapter and that was implemented on the TMS320C50 has the interface shown in Figure 6.1; the algorithm was devised by the engineers in Texas Instruments Italy.

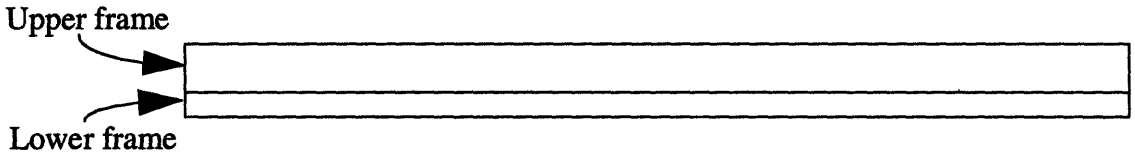


Figure 6.1 Unboxed recognition

Although the writing area is still divided into an upper and lower frame, the dividing lines that create boxes for the characters are no longer present; thus, characters, when written, may now be spaced either closer or farther apart than in the boxed recognition system. The characters still, however, must be written discretely, meaning that the characters must not be touching. These type of characters are called spaced, discrete characters, as mentioned in Chapter 1. Segmentation still must be performed from this type of writing input, but it is facilitated by the fact that segmentation is performed between strokes, not within strokes as in the case of cursive script recognition.

This unboxed environment provides more flexibility in how the characters are written. Although the characters must not be touching, they may still be much closer than the boxed recognition environment allows. For example, when printing, it is often very natural to have character boxes overlap such as in Figure 6.2, where the character box of the 'e' crosses the character box of the 'T'. The character box is the box that is created using the minimum and maximum coordinates of the character as corners of the box, as described in Chapter 3. This algorithm allows the proper recognition of these characters such as the 'T' and 'e' despite the fact that their character boxes overlap.

Figure 6.2 Overlapping character boxes allowed in unboxed recognition

Issues that arise from this unboxed recognition algorithm include proper separation of strokes and proper recognition of spaces between words and characters. For instance, it is now necessary to determine whether the spacing between two characters is representative of an empty space between the characters and hence a separation between two words or if that spacing is merely a space between two consecutive characters in the same word. Clearly, this type of recognition, though more flexible, is more complicated and possibly more susceptible to errors in recognition.

From Figure 6.1, it is apparent that there are still some constraints on the input characters. Although the width of a character is no longer limited, the height is still limited by the unboxed environment. Furthermore, the frames in the writing area are still used, thus enabling the easy division of characters with and without descenders. This type of unboxed environment with these constraints is very common and still provides much freedom, and it is exactly the same as the environment in ruled paper. This more natural unboxed input environment is a great improvement over the less natural boxed environment.

This chapter will discuss the unboxed algorithm and describe its implementation on the DSP chip. Section 1.2 will provide a description of the basic operation of the algorithm with respect to segmentation and proper recognition of the segmented characters. Section 1.3 will next describe the interface of the unboxed recognition algorithm and the implementation of the algorithm on the DSP chip. Section 1.4 will then provide the results from profiling the implemented unboxed recognition code on the DSP chip, and Section 1.5 will give some final comments on this algorithm and implementation.

6.2 The Algorithm

As mentioned above, the unboxed recognition algorithm recognizes spaced, discrete

characters; it thus requires that the characters not be touching and be completely separated. The unboxed recognition algorithm uses much of the same functions and performs many of the same steps as in the boxed recognition algorithm. The preprocessing of the characters is identical to the boxed recognition algorithm, and the recognition steps are almost also identical. Figure 6.3 shows the steps involved in unboxed recognition.

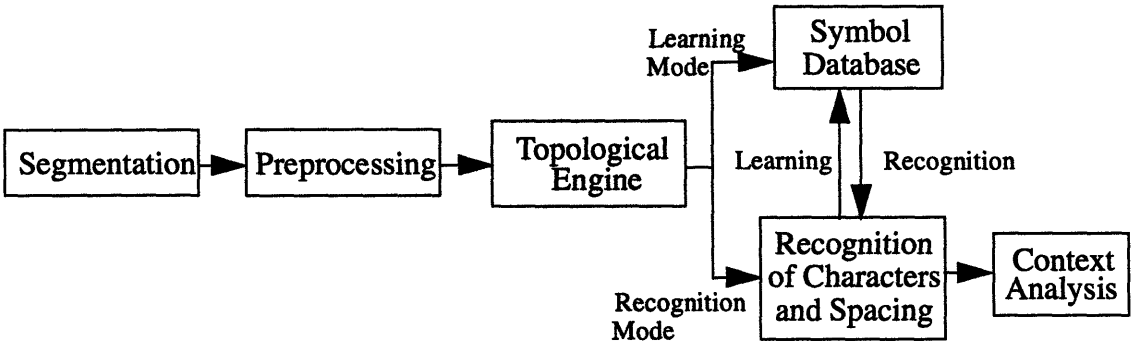


Figure 6.3 Unboxed recognition algorithm flow diagram

The first step in the unboxed recognition algorithm is the segmentation of characters and the proper allocation of strokes to the correct character. The purpose of this step is to create character boxes so that the input to the preprocessing stage is identical to that in boxed recognition. Each character written in the unboxed environment is properly segmented and the strokes are gathered so that each character is virtually boxed, or segmented. During the gathering of strokes, the segmentation sometimes suffers for characters with disconnected and non-touching strokes, but these problems are solved in the recognition stage. After segmentation, the characters are then sent to the preprocessing unit to be cleaned and normalized, and the preprocessing functions are the same as those described in Chapter 2. The appropriate dynamic information and topological features are extracted, and then the characters are sent to the recognition unit. The recognition unit performs recognition in the same way described in Chapter 2, with scores being calculated for the similarity between the input character and each character from the symbol database. The character in the database with the lowest score is the recognized character. There are, however, additional steps in the recognition phase, with the proper recognition of spaces and with context analysis, and these steps are used to resolve such unboxed input issues as spacing between characters and correct recognition of characters that might have been improperly segmented. The following subsections will describe each stage of the unboxed recognition algorithm in more detail.

6.2.1 Segmentation

The segmentation phase of unboxed character recognition is designed to separate the strokes and associate each stroke with a character. The goal is to have the characters in the same format as the format for boxed recognition; thus, after the segmentation stage, the

characters should appear as if they were written in the boxed environment. As mentioned in Chapter 1, the segmentation of these spaced, discrete characters is facilitated by the fact that segmentation is performed between strokes, not within strokes as in cursive script recognition. The segmentation phase is still, however, a non trivial process in determining the correct association of each stroke to the correct character. The segmentation process occurs in several stages to ensure proper gathering of the strokes.

The first step in collecting the strokes is to spatially sort the strokes in the stroke array variable. The strokes, when originally written by the user, may be written in any temporal order. Thus, a character need not be completed before the next character is written. This flexibility allows users to, for example, dot the i's and cross the t's at the end of a word rather than at the time that the letter is first begun. The user may choose to incompletely write a character at one time and complete the final strokes of the character at a later time. Because of this temporal freedom in writing all characters, in order to facilitate the segmentation process, it is necessary to sort these strokes spatially so that strokes that are next to each other geometrically are located next to each other in the stroke array variable. The result of geometrical sorting is shown in Figure 6.4. Figure 6.4(a) shows the original order of the strokes, as written by the user, and Figure 6.4(b) shows the sorted order of the strokes in the stroke array. As the figure shows, the sorting of strokes is performed in the positive x-direction, i.e., from left to right.

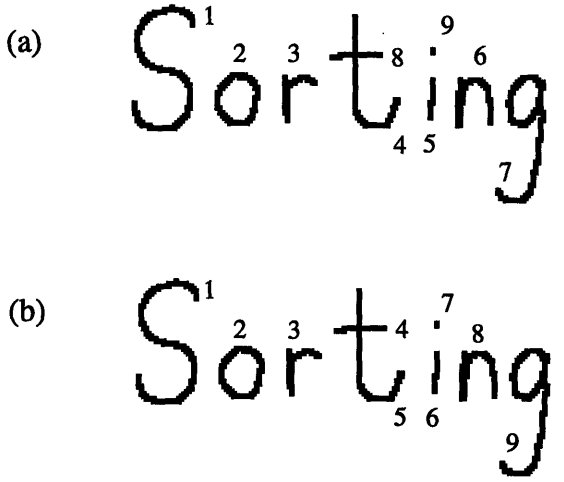


Figure 6.4 Geometrical sorting of strokes (a) Original order (b) Sorted order

After sorting the strokes, the actual segmentation process can begin to determine if strokes that are nearby one another belong to the same character or to different characters. This process begins by examining two consecutive strokes in the stroke array variable, $stroke_i$ and $stroke_{i+1}$; these two strokes are spatially next to one another. Tests are then performed to determine whether or not these strokes belong to the same character. If they do belong to the same character, they are merged into the current character box, and $stroke_{i+1}$ and $stroke_{i+2}$ can be compared next to see if $stroke_{i+2}$ belongs to this character and needs to be

merged into this current character box. This process continues until it is determined that two consecutive strokes do not belong to the same character. When this occurs, then the current character box is considered complete, meaning that all of the strokes that belong to the character are considered to be merged together to form the current character box. Thus, if it is found, for example, that $stroke_{i+1}$ and $stroke_{i+2}$ do not belong to the same character, then the character box that $stroke_{i+1}$ belongs to is complete. Then, the entire process begins again, and $stroke_{i+2}$ and $stroke_{i+3}$ are examined to see if they belong together in the next character box.

In order to determine whether the strokes, $stroke_i$ and $stroke_{i+1}$, belong to the same character or to different characters, the basic idea is to use the distance between the two strokes as the deciding factor. If two strokes are intersecting or if they are within a delta of one another, then the two strokes may be considered to be part of the same character. The exact details of the segmentation routine are not included, as the algorithm is considered by Texas Instruments to be proprietary. This type of stroke gathering has a couple of implications. The calculation of the delta must be done very carefully; the delta must be large enough to allow the proper gathering of strokes that are disconnected but belong to the same character. It is desired, however, to have the delta small so as to not incorrectly gather strokes that do not belong to the same character. The issue of poorly chosen deltas is shown in Figure 6.5.

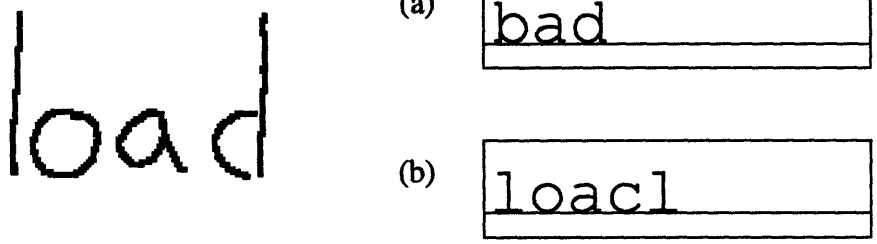


Figure 6.5 The word 'load' is written. (a) delta is too big (b) delta is too small

Another issue that arises is that this type of segmentation requires that the characters not be touching. Furthermore, the characters cannot even be too close to one another or then they will be misrecognized. They must at least be delta apart, and this further necessitates the need to keep delta small so as not to constrain the user too much. In order to achieve high accuracy, the user must follow these constraints when writing the characters in the unboxed environment.

Thus, after all of this stroke gathering, the result is a character box for each character. Figure 6.6 illustrates an example of character boxes constructed for characters written in the unboxed environment. These character boxes are then ready to be spent to the preprocessing stage, which is the next step in unboxed recognition and is discussed in the next subsection.

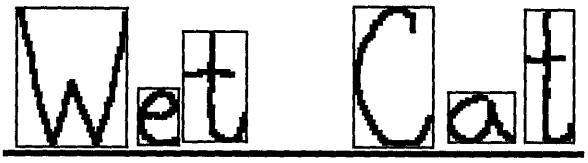


Figure 6.6 Character boxes for characters written in unboxed environment

6.2.2 Preprocessing and Topological Engine

The output of the segmentation phase is the character box of each character; this output is identical to that produced in the boxed recognition phase after writing each character in a box. The purpose of the segmentation phase was to standardize the input so that the unboxed recognition algorithm could use the same preprocessing and topological engine functions as the boxed recognition algorithm. Thus, the preprocessing and topological engine functions are the same as those described in Chapter 3. The preprocessing functions consist of a function for scaling and interpolation of the input character and one for thinning the character. After all of this preprocessing work, the output is the standardized character matrix that is used in the topological information extraction. The topological engine functions include one for feature extraction, one for microfeature detection, and one for finding the dynamic information. From the topological engine stage, the result is the information that is used in the recognition functions described in the next subsection.

6.2.3 Recognition of Spacing and Characters

The recognition functions take as input the results from the preprocessing and topological engine stages. Many of the recognition functions are the same as in the boxed recognition algorithm. There are, however, a couple of additional steps that are designed to take care of the problems introduced by unboxed recognition. Specifically, special functions are designed to ensure proper recognition of spaces and the proper recognition of specific characters that consist of unconnected strokes.

The first step in the recognition of unboxed characters, after all of the segmentation, preprocessing, and topological engine extraction are completed, is to properly recognize spaces among the characters written. The spacing function used in this unboxed recognition algorithm is designed to take into account the fact that some people write characters in a word farther apart than other people. Thus, the determination of spaces is dependent on how the user writes characters. If a user writes characters far apart in a typical word, then it would take a large gap between characters to be recognized as a space. If the user, on the other hand, writes characters close together in a word, then it takes a smaller gap to be recognized as an empty space.

The goal is to calculate the average spacing of all of the written characters and use this average in a thresholding function to determine where spaces are located. The use of the

average is a good indicator for where spaces are located if appropriate outliers are eliminated from the calculation. That is, the average would be very misleading if characters that have very small spacing between them or characters that have very large spacing between them are used. Thus, all spaces that do not fall within a certain range are not used in the calculation of the average. The average spacing between all characters that satisfy the above criterion is then used in a formula to obtain the final threshold, and spaces between each character box are compared to this threshold to determine whether or not an empty space should be inserted. If the space between two character boxes is less than the threshold, no space is inserted and the two characters are considered to belong to the same word; otherwise, an empty space is inserted between the two characters, and thus the two characters are considered to belong to different words. The exact details of the spacing function are not included, as the algorithm is considered proprietary by Texas Instruments.

After the characters are examined for spaces, the next step is to perform the character recognition. The characters are recognized the same way as described in Chapter 3 for boxed recognition. The original recognition function for boxed recognition simply takes the input character and each character in the symbol database and calculates a score based on the similarity between the two in features, microfeatures, dynamic information, height and width, and number of strokes. The character from the symbol database with the lowest score is used as the recognized character. After all of the unboxed characters are recognized, the next step is to properly recognize those characters that might have been incorrectly segmented. This step is called context analysis, and it will be discussed in the next subsection.

6.2.4 Context Analysis

This is the final step in the recognition process of unboxed characters. It is designed specifically to identify and recognize those characters that could be incorrectly segmented by the segmentation phase. These characters consist of strokes that do not touch or intersect and that could thus possibly create problems during segmentation. They also consist of strokes that may easily be merged, such as in the percent sign, and strokes that are points, such as in 'i' or 'j', whose point may be too big or too long to have been properly identified during the segmentation stage. The characters that are examined and searched for are shown in Table 6.1.

Table 6.1 Characters recognized by context analysis

"
=
i
j
%

Each of these characters is not always properly segmented by the segmentation routine, so special context analysis functions are used to ensure their proper recognition. The goal is to identify consecutive characters, among the characters that were recognized in the previous stage, that may belong to one of the characters in Table 6.1. For example, with the quotation mark, the search is for two consecutive apostrophes. Once these characters are identified, then tests are performed to properly determine whether the characters actually should be merged into one character or whether the character should be left separately, thus indicating that the segmentation was indeed performed correctly. The exact details of the context analysis routines are not included, as the algorithm is considered proprietary by Texas Instruments.

After all of the context analysis, the recognized characters are examined, and if any of the characters need feedback, the character is recognized again with greater emphasis placed on the character's dimensions. The same feedback table as shown in Table 3.1 is used to properly recognize those characters that differ only in height and width but have the same shape. After the feedback, then the recognition of unboxed characters is considered complete.

6.3 Interface and Implementation on the TMS320C50 DSP Chip

6.3.1 Interface

The unboxed recognition algorithm interface is very similar to that of the boxed recognition implementation. The user has the option to write characters with the WRITE option, teach the recognition system a writer's style with the LEARN option, select a user's personalized symbol database with the USER option, and insert a misrecognized character into the symbol database with the INSERT option. The writing interface for the unboxed recognition implementation is shown in Figure 6.1. As in the boxed recognition interface described in Chapter 4, the writer in this unboxed environment has the ability to delete the last stroke written with the BACK option and delete a specified character with the DELETE option. The learning for unboxed recognition is performed in the boxed environment, as the segmentation stage standardizes a writer's input in the unboxed environment to make it appear as if it were written in the boxed environment. Thus, the learning can be performed in the boxed environment, and the created symbol database can then be used to recognize unboxed input. The learning interface is exactly the same as that in the boxed recognition interface. Likewise, the user interface to select the symbol database is identical to the boxed case.

The primary difference between the boxed recognition interface and the unboxed recognition interface lies in the interface for the insertion of characters. The insertion of characters in the unboxed case is important, especially because of the possibility that the recognition algorithm can make mistakes in the segmentation of characters. The INSERT option allows the user to collect all of the strokes in a character and identify the character with the proper symbol. The insertion of a character in the unboxed environment is shown in Figure 6.7.

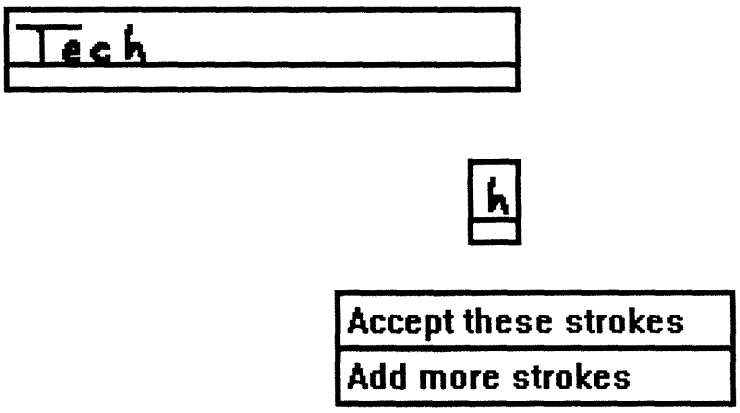


Figure 6.7 Insertion in the unboxed recognition implementation

Thus, if a character is improperly segmented, the user can adjust the gathering of strokes until all of the strokes of the character are collected. The user then selects the correct identifier, once the strokes are gathered, in exactly the same manner as in the boxed recognition interface.

6.3.2 Implementation

The first step in the implementation of the unboxed recognition algorithm on the TMS320C50 DSP chip is to determine which functions the PC will perform and which functions the DSP chip will perform. This step is necessary to ensure that the DSP chip performs only those tasks for which it is well-suited, and this careful consideration will result in the recognition process operating as fast as possible. As in the boxed algorithm, it is a logical division of labor to have the PC perform the input/output tasks and graphics jobs, while the DSP performs the actual work of the preprocessing of characters and recognition of characters. The additional steps to the boxed recognition for unboxed recognition are segmentation, recognition of spaces, and context analysis. It is decided to allow the PC to perform the segmentation and the DSP to perform the space recognition and context analysis. This decision was based on the fact that the segmentation of characters is more of an input/output task, and it can be more efficiently performed on the PC. The context analysis and space recognition are more suited for the DSP because of their data intensive nature.

Recall that another factor in determining which sections of code should be processed on the PC and which should be processed on the DSP is the number of data transfers required between the PC and the DSP. The division of having the segmentation performed by the PC and the context analysis and space recognition on the DSP minimizes these very time-consuming data transfers. Thus, when a user writes characters on the digitizing tablet, the PC first gathers the points and puts the data into the appropriate variables. Then, the strokes are segmented on the PC and the character boxes are constructed. Then, as in the boxed recognition implementation, character boxes are sent to the DSP for preprocessing and recognition. This choice of division of labor allows a minimized and efficient number

of data transfers. With the space recognition and context analysis, much of the data that is required is the result of the preprocessing stage that is performed on the DSP. Thus, it would be very inefficient to have the PC perform the space recognition and context analysis and thus require that the data be transferred to the PC for these two functions and back to the DSP for recognition. Figure 6.8 illustrates the division of labor chosen for this implementation.

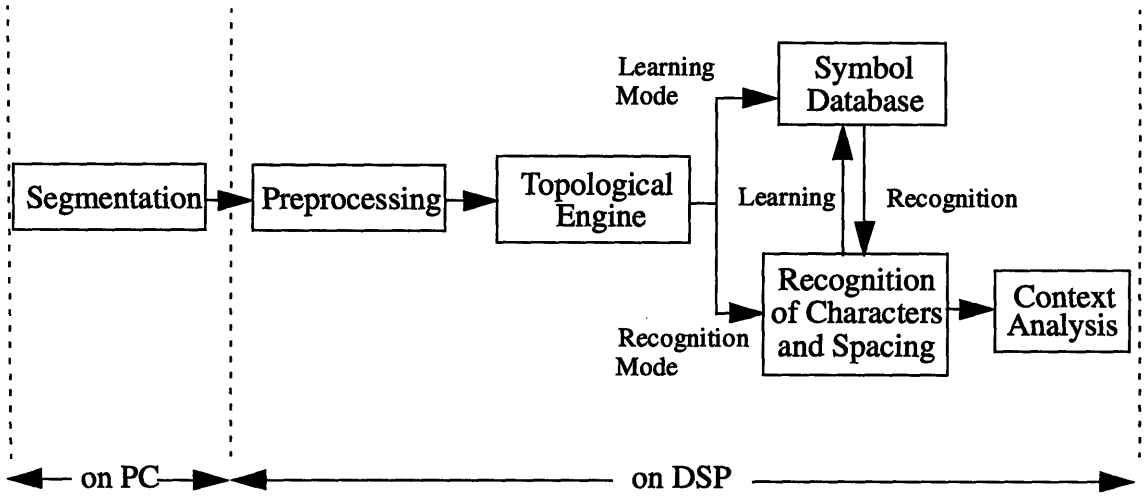


Figure 6.8 Division of labor chosen for unboxed recognition

The implementation of the unboxed recognition algorithm on the TMS320C50 requires that the code and data space fit within the confines of the memory provided by the EVM and DSP chip. Recall that the EVM and DSP chip allow 64K 16-bit words of external memory and 9K 16-bit words of on-chip memory to be used. The code size of the unboxed recognition algorithm is 11482 words, and the data size necessary, including the size for the strokes information, points information, and symbol database, is 33228 words. In addition to the code and data space, 8192 words are allocated for the stack, which holds the local variables, and 1280 words are allocated for the heap, which holds the dynamic memory. In order to minimize the data accessing conflicts described in Chapter 5, much of the code is placed in the on-chip memory, while the data is placed in the external memory.

In order to implement the algorithm on the DSP chip, it is essential to set up the data transfers so that the necessary data is sent to and from the DSP. The data must be sent 16 bits at a time. The goal is, while sending all of the necessary data, to minimize these time consuming data transfers. The unboxed recognition algorithm has the same data constraints as described in Chapter 4. Each character stored in the database is 86 bytes. When sending this data for each character, as in the boxed recognition case, it is necessary to compact bit fields. This is helpful in sending the data the least number of times necessary. After compacting the bit fields of the features, microfeatures, and dynamic information into 16-bit words, the first step is to simulate the data transfers on the PC to

ensure that the data transfers are complete and that the appropriate data is sent and received. When performing these simulations, the purpose is to ensure that each function on the DSP and each function on the PC has the correct information necessary to perform its task. After the simulations, the next step is to use the EVM communications interface described in Chapter 4 to send the data to and from the DSP chip. This interface allows the EVM and PC to communicate properly and to follow the handshaking protocol described in Chapter 2. The appropriate commands need to be created for the PC to tell the EVM what to do, and the same commands are used in the unboxed recognition implementation as in the boxed recognition implementation as shown in Table 4.2, with the addition of one command, GET_REMAKE_DATA. This command is used when updating the database after the insertion of a character or the learning of characters, and it eliminates unnecessary data transfers that occurred in the boxed recognition implementation with only the GET_DATABASE command. After the appropriate commands are created, then the code is loaded into the EVM and into the PC, and the algorithm can be run. So, after the data transfers are completed using the same communication interface as described in Chapter 4, the algorithm can be effectively run on the PC and on the DSP chip. Figure 6.9 illustrates the steps involved in preprocessing, and it shows both the PC's and the DSP chip's roles in the process.

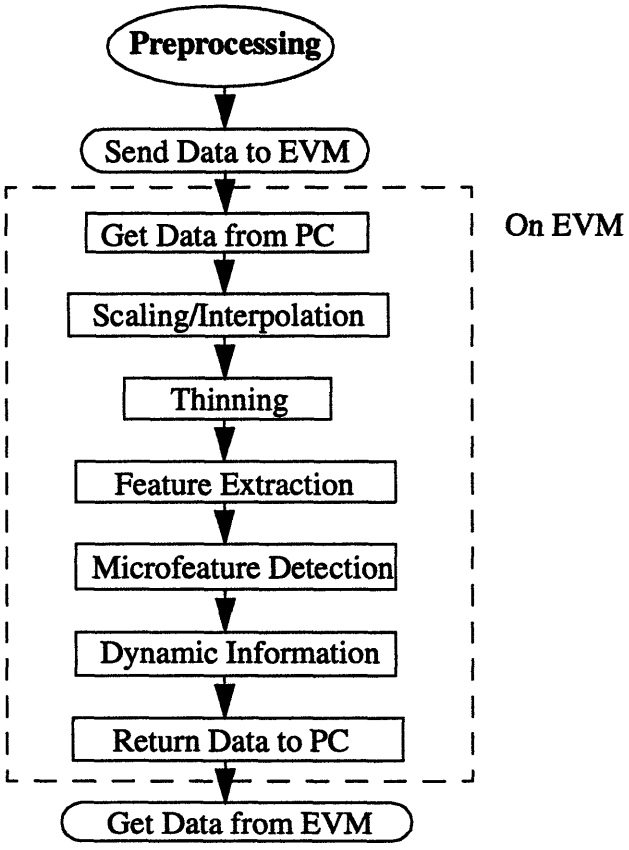


Figure 6.9 Preprocessing stages on PC and DSP

Figure 6.10 is a similar illustration of the recognition process.

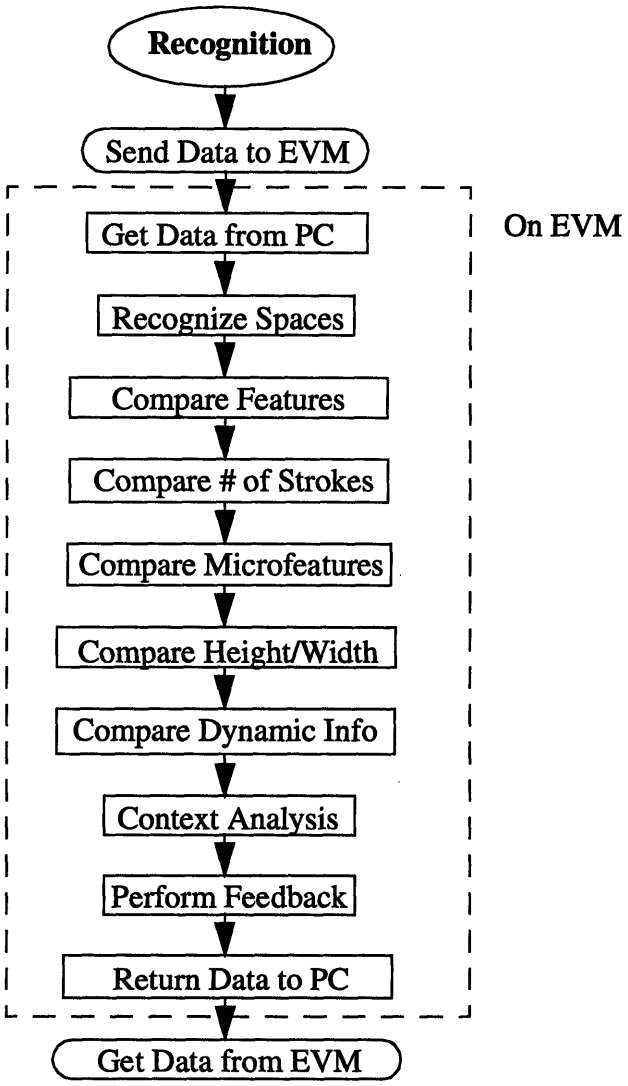


Figure 6.10 Recognition stages on PC and DSP

6.4 Results

The unboxed recognition algorithm achieves a very high accuracy rate while allowing the user much more freedom in input in writing the characters. Tests performed to check the accuracy of the unboxed algorithm yield a lower accuracy rate than for the boxed algorithm, but the accuracy still remains very high. This drop in accuracy is due primarily to the difficulty in segmentation and the difficulty in the user's writing characters without having them accidentally touching or merged. After being implemented on the DSP chip, the next step, analogous to that performed in Chapter 4 for the boxed recognition implementation, is to profile the code to determine the speed of recognition and the

location of time consuming kernels. The actual speed of recognition is dependent on the size of the database; the greater the number of characters in the symbol database, the more the number of comparisons that have to be made during recognition and thus the more time spent during the recognition stage. In the unboxed recognition case, more characters, compared to in the boxed recognition implementation, are stored in the symbol database to improve the recognition accuracy, and for this reason, the profiling results in the unboxed case cannot be compared to the results in the boxed case. Instead, these profile results can be used to determine the relatively time consuming stages in the recognition of unboxed characters, and this result can be helpful in determining where optimizations can and should be performed. Table 6.2 below shows the relative time spent in preprocessing and recognition.

Table 6.2 Relative Time Spent in Recognition and Preprocessing

Function	Relative Time Spent
Preprocessing	4.4%
Recognition	95.6%

As Table 6.2 indicates, much of the time is spent in the recognition stage rather than in the preprocessing stage. Table 6.3 shows the relative time spent in each function in the preprocessing stage, and finally, Table 6.4 shows the relative amount of time spent in the time-consuming recognition stage.

Table 6.3 Relative Time Spent in Functions Within Preprocessing

Preprocessing Function	Relative Time Spent
Preprocessing	100%
Scaling and Interpolation	26.4%
Thinning	14.7%
Feature extraction	33.4%
Dynamic Information	11.0%
Microfeature Detection	14.5%
Other	~0.0%

Table 6.4 Relative Time Spent in Functions within Recognition

Recognition Function	Relative Time Spent
Recognition	100%
Comparing Dynamic Information	37.1%
Comparing Microfeatures	33.5%
Comparing Features	26.5%
Other	2.9%

The total time to recognize a single character, from profiling the preprocessing and recognition of many characters, is about 65 ms, using a 25ns TMS320C5x DSP chip. This time, again, cannot be compared to the boxed case due to the different number of characters in the database.

6.5 Conclusion

The unboxed recognition algorithm and implementation presented in this chapter allows the user to have much more freedom in the writing input. The writing input is no longer limited to boxes, but now the writer may write the characters in a more unconstrained environment by having the flexibility to write the characters with varying spaces between characters. This improved environment performs the recognition of spaced, discrete characters.

The recognition operates by segmenting the characters to determine the ownership of each stroke, and after segmentation, the preprocessing and recognition occur as before, with the added steps of space recognition and context analysis. Space recognition detects the location of spaces among the characters, and the context analysis facilitates the recognition of those characters that are difficult to segment. These additional functions, while providing the ability to make the algorithm more flexible in terms of the input, do not add too much complexity or size to affect the compactness of the algorithm code, and the recognition accuracy for unboxed characters is still very high. The speed of the algorithm is slower than in the boxed case, but the speed should increase dramatically once similar optimizations are performed on the unboxed recognition functions as were done on the boxed recognition functions.

The accuracy of this recognition is very high, but is slightly lower than for boxed recognition. It suffers from problems with the segmentation of characters, which affect its recognition. Often times, with unconstrained writing, it is easy to become careless and accidentally have two characters touching. If the characters touch, though, then the

segmentation is severely affected and is not accurate. Another problem occurs when the spacing between characters written by the user is not consistent. In this case, unwanted spaces or not enough spaces will be added by the recognition system, and thus the output of the recognition system would be incorrect.

This implementation was optimized by the C compiler but not optimized by hand. The profile results clearly show that some improvement in recognition speed can be achieved through further optimizations. Because many of the functions between the boxed and unboxed recognition are similar, it is possible that the same or very similar rewritten assembly functions can be used to optimize the recognition in terms of speed. Thus, the next step could be to optimize the unboxed recognition code to achieve an even faster recognition rate.

Chapter 7

Conclusion

7.1 Overview

This thesis describes the real-time implementation of two character recognition algorithms on Texas Instruments' TMS320C50 DSP chip. The implementation is designed so that the DSP chip can perform the more data intensive and thus more time consuming functions, while the PC performs the graphics tasks and input/output interface with the digitizing tablet. These tasks that the PC performs are those that the DSP cannot handle very well; the DSP is not able to perform as many tasks as a microprocessor, but those tasks that the DSP chip can perform are executed at a much faster rate than when using a standard microprocessor. The DSP chip contains an efficient, yet limited, instruction set as well as possessing the ability to handle data in a minimum number of cycles, and these factors contribute to the effectiveness of the DSP chip. Thus, the DSP is the main engine for performing character preprocessing and character recognition, and this results in a significant increase in recognition speed.

The two algorithms that were implemented are one that performs boxed recognition and one that performs unboxed recognition. The boxed recognition algorithm recognizes, in real-time, characters that are each written in an individual box. The algorithm is user-trainable, so each user using the recognition system may teach the recognizer a specific writing style by creating a personal symbol database, which significantly increases the recognition accuracy. Those characters that are not recognized correctly can be corrected immediately by the user, and the proper identifier will be inserted into the symbol database. Each character, before being recognized, first undergoes preprocessing. This step is designed to clean and standardize the input for the next stage. The goal is to eliminate any unwanted noise or any irregularities in the character caused by peculiarities in a user's writing style. The input character is first scaled and interpolated; this step standardizes the input character's size along with either adding points to or deleting points from the character. The points need to be added to the input character when the character is quickly written; in this case, the digitizing tablet does not get to sample many points in the character while it is written, so the preprocessing stage adds the points through an interpolation routine. On the other hand, if the character is written very slowly, then many points are sampled by the tablet, so the preprocessing unit eliminates any redundant points. At the end of the scaling and interpolation, the output is a scaled character matrix. This matrix is then subjected to the second and final phase of preprocessing, thinning. The goal of thinning is to eliminate unwanted or unnecessary points in the character matrix. These are points that can be eliminated because they do not add any more meaning to the character, and when removed, the character can still be uniquely distinguished from any other character in the database.

After the preprocessing stage, the standardized character matrix is then sent to the

topological engine stage. In this section, appropriate information that will be used in the recognition engine is extracted from the input character matrix as well as from the points in the character. The information obtained from the character matrix includes the features and microfeatures, which are optical information that will help to identify both the relative and absolute spatial positions of the points. The third type of data is extracted from the points collected when the digitizing tablet samples the input strokes as they are written with the stylus. This data is the dynamic information, and it describes the relative temporal locations of point in the character. After all of this topological information is collected, the next and final step is to send this information to the recognition stage.

The recognition stage takes the topological information of the input character and examines one input character's data at a time. The data is compared with the data compiled in the symbol database. Each character in the symbol database is compared with the input character with respect to character height and width, number of strokes, features, microfeatures, and dynamic information. A score is calculated based on the similarity of the two characters, and a lower score indicates a greater degree of similarity. After all characters in the symbol database have been inspected and compared with the input character, then the character with the lowest score is selected as the correct identifier. This process continues for all of the input characters, and the result is that each written character has an appropriate identifier. The final step in the recognition stage is to examine these recognized characters and to correctly recognize those characters that are similar to others, such as 'c' and 'C', and that differ only in height or width. This is accomplished by placing more emphasis on the height and width scores of the recognition functions. After this final feedback, then the recognition process is complete. The boxed recognition algorithm recognizes characters with great success. The database can be created by training the recognition system, and it can be personalized to recognize various sort of styles for the same character. Furthermore, a character can be inserted into the database a multiple number of times, which increases the likelihood that a character will be correctly identified anytime when written. With the boxes to separate characters and frames to help divide characters, the boxed recognition algorithm can achieve a very high recognition accuracy rate.

The boxed recognition is implemented on the TMS320C50. The DSP chip is located on a platform to allow full simulation and testing of the software using the TMS320C50. The platform used is the Evaluation Module (EVM) board. The DSP chip is capable of performing the preprocessing and recognition, while the PC performs the interfacing tasks. The PC and DSP communicate through a communications protocol that allows the PC to send commands to the DSP to tell it what to do and allows the PC to send the appropriate data to the DSP, 16 bits at a time. Because of this limitation in sending the data only 16 bits at a time, the goal is to minimize the time consuming data transfers. Furthermore, because of the constraints provided by the 16-bit data bus, some of the data being sent must be modified in order to properly send and receive the data. Once the appropriate changes to the data are made, then the DSP and PC can communicate effectively to run the algorithm smoothly.

In the typical process, the user first writes characters on the digitizing tablet. As the characters are being written, each point that is sampled by the tablet is stored in the appropriate structure variable on the PC. Similarly, the number of strokes and number of points per stroke are also stored; once all of the characters are written and the user wishes to have them recognized by the system, the appropriate data is sent to the DSP, 16 bits at a time. The DSP chip then takes the received data from the PC and first performs the preprocessing. Then, the topological data and dynamic information are extracted, and this data is then ready for recognition. Once the DSP performs the recognition of all of the input characters, then it sends only the necessary information, such as the identifier, back to the PC. The PC then takes the identifier and appropriately displays the correctly recognized characters on the digitizing tablet and on the screen. All of the above processes occur with the PC initiating and ending communication. The PC drives the DSP and tells the DSP exactly which data to receive, what to do with the data, and which data to return. This kind of communications protocol ensures that a loss in synchronization will never occur between the PC and DSP chip.

The boxed recognition algorithm is written entirely in C. In order to run the algorithm on the DSP chip, the code must first be compiled using the TMS320C5x DSP compiler, and this translates the code to the assembly language used by the DSP chip. The assembly files created are then assembled and converted to object files that can be linked to form a final executable program. During the compiling process, the optimization option can be turned on to allow the compiler to perform optimizations in the translation of C to assembly, and this will yield code that runs significantly faster than if the option were not used. These improvements, however, are not fully optimal, and even greater reductions in cycle times can be achieved by rewriting the assembly programs generated by the optimized compiler.

The next step is thus to write specific assembly programs by hand that can be used to significantly reduce the number of cycles necessary to perform tasks. The time-consuming kernels are identified by profiling the code on the DSP. The profiler allows the number of cycles that each function on the DSP performs during the recognition of characters to be calculated, and from this result, the time consuming functions can be easily determined. Thus, time consuming functions are rewritten in assembly to save cycles and reduce the total recognition time. The optimizations are performed by using the special characteristics and special instructions of the DSP chip. Optimizations performed include the use of more registers, more effective use of the stack, more efficient variable accessing, and the use of on-chip memory to reduce data accessing and code accessing conflicts. The optimizations are performed on three recognition functions, two preprocessing functions, and three topological engine functions. These optimized functions result in a very significant reduction in the amount of time the system takes to recognize a single character. With a DSP chip with a 25ns instruction execution time, the average time to recognize a single character is reduced from about 15 ms on the DSP to about 6 ms per character on the TMS320C50.

The second algorithm that is implemented in real-time on the DSP chip is one with a more

flexible input environment; it recognizes hand printed characters in an unboxed environment. The characters must be spaced and discrete; the process of recognizing characters is very similar to the boxed recognition algorithm, with the addition of three stages. The first step in the recognition of unboxed characters is the segmentation stage, and this step is performed immediately after all of the characters are written. The segmentation process is designed to separate the characters and to correctly allocate the strokes to the correct characters. The result of this stage is that each character will be put in a character box, and thus the input to the preprocessing stage is identical in the unboxed case as in the boxed case. Thus, the output after segmentation should make the unboxed characters appear as if they were written in the boxed environment. After the segmentation, the next step is to perform preprocessing in the same manner as in the boxed recognition case; after the preprocessing, then the topological information is extracted in the same way as in the boxed case. Finally, the characters are ready for recognition. The recognition stage contains two additional steps to those in the boxed recognition implementation. The first is the process of recognizing spaces written among the characters. In the boxed environment, there is no confusion as to where spaces are located; in the unboxed environment, however, there can be much confusion in detecting spaces. Thus, space recognition is performed to identify when gaps between characters are actual empty spaces between two words and when gaps are merely the normal spaces between characters of the same word. After the space recognition, the characters are recognized using the same functions as in the boxed case. Again, the input character is compared with all of the characters in the database, and the appropriate identifier is extracted. The next step in the unboxed recognition process is context analysis. Context analysis is designed to properly recognize those characters that consist of unconnected strokes or merged strokes that were incorrectly segmented and that might have thus been incorrectly identified. Special examination of the recognized characters is performed to identify potentially misrecognized characters, and if necessary, characters are merged and correctly identified. Finally, after context analysis, feedback is performed on all of the recognized characters to eliminate the incorrect identification of characters that are written with the same stroke sequence and differ only in height or width.

As in the boxed recognition implementation, the unboxed character recognition algorithm achieves a very high accuracy rate. Because the purpose of the segmentation process is to make the characters written in the unboxed environment appear as if they were written in the boxed environment, then the recognition, if it is incorrect, usually fails due to improper segmentation. The segmentation stage requires that the input characters all be spaced apart and not touching. If this does not happen, then the results will be incorrect and separate characters will be merged during the segmentation phase and thus misrecognized. Another common problem occurs with the detection of spaces. The space recognition requires that spacing among characters be consistent, and if characters within a word are too far apart, a space may inadvertently be inserted. Similarly, if spaces between words are not large enough, then words may be merged into a single word. Despite these two problems, the careful inputting of characters will result in a very high accuracy rate, and as in the boxed case, the symbol database may be customized in order to eliminate the effect of a user's variance in writing the same character.

The unboxed recognition algorithm is implemented on the TMS320C50. Again, the EVM is used as the platform to allow full testing of the software on the DSP. The first step involves determining the most logical division of labor between the DSP chip and the PC. As in the boxed case, the PC will perform the graphics and input/output tasks, while the DSP chip will perform the preprocessing and recognition. Of the three new steps introduced by unboxed recognition, the PC performs the segmentation and the DSP performs the recognition of spaces and context analysis. This decision allows each to perform its optimal type of task, and the number of time consuming data transfers is minimized. Only the essential information required by the DSP chip is transferred to the EVM, and likewise, only the essential output from the DSP chip is given back to the PC. Setting up the data transfers requires a complete understanding of the functions and requires absolute assurance that all of the necessary data is sent. Thus, simulations are first performed on the PC to ensure the data transfers are correct. Then, using the same communications protocol as in the boxed recognition case, the data can be actually transferred between the PC and the DSP, and the DSP chip can correctly run its code. Many of the same data constraints as in the boxed case exist in the unboxed case, so many of the same changes have to be made to allow the data to be optimally sent 16 bits at a time. The on-chip memory is used in this implementation to reduce the number of cycles wasted in data conflicts. As in the boxed recognition implementation, the next step after implementation is to profile the results and identify the time consuming kernels. It is shown that because of the large symbol database used in the unboxed recognition process, the recognition stage is much more time consuming than the preprocessing stage. Time consuming functions are identified, and the next step can be to optimize those functions in assembly. However, optimizations are not performed in this thesis. With the extremely large database and with the only optimizations being performed by the compiler, the recognition time per character is about 65 ms, using a DSP chip with a 25ns instruction execution time. Significant reduction of cycles can be achieved with proper optimizations in an analogous manner to the boxed recognition case.

7.2 Improvements and Future Work

One possible improvement that can be made to both of these real-time implementations of a character recognition algorithm on the DSP chip is to use a dictionary to implement one of the techniques described in the Hidden Markov Model approach in Chapter 1. In this approach, when characters in a word are being identified, rather than each character having just one possible identifier, there is often more than one identifier per character. This idea can be transferred to the recognition algorithms described in this thesis; if multiple characters in the database receive low scores when being compared to an input character, which imply that the characters from the database have a high degree of similarity to the input character, then these characters can all be retained as possible identifiers for that input character. Thus, if the score of a character from the database satisfies some sort of threshold criterion, such as

$$score < \delta_{similarity}$$

then the character is kept as a possible identifier. The threshold equation can be more complex than the above one, such as rather than being an absolute threshold, it can be a function of the lowest score or of a multiple number of scores. Or, perhaps, it can be a function of the variance of all other scores from the lowest score. The selection of the proper threshold equation is user-dependent. If the absolute threshold formula shown above is used, the threshold can be user-defined to allow the retainment of many identifiers for each input character, or to allow only confusing input characters to retain more than one identifier.

During the recognition of an input character, each score of the characters from the symbol database can have the threshold criterion applied to it. Then, once all of the possible identifiers are gathered for each character, then a dictionary can be used to properly determine the correct recognized word from all of the possible combinations of characters. If there are multiple definitions of words, then in the worst case, the word can be identified in the same manner as with the algorithms in this thesis, using only the identifiers with the lowest scores. But, often times, there is just one or two confusing letters in a word, such as in Figure 7.1, and this method would resolve this problem and correctly identify the word. In the figure, the word is originally recognized as 'Lar'. Because the first input character has two possible identifiers, 'C' and 'L', the dictionary will be used to determine the correct word. The two possible identifiers are then used, one at a time, in the word and sent to the dictionary. Thus, the words 'Lar' and 'Car' are sent to the dictionary, and the dictionary then selects 'Car' as the appropriate word. The dictionary is used as kind of a spell checker, and the possible identifiers of the input characters serve as the deciding factors for selecting the correctly spelled word.

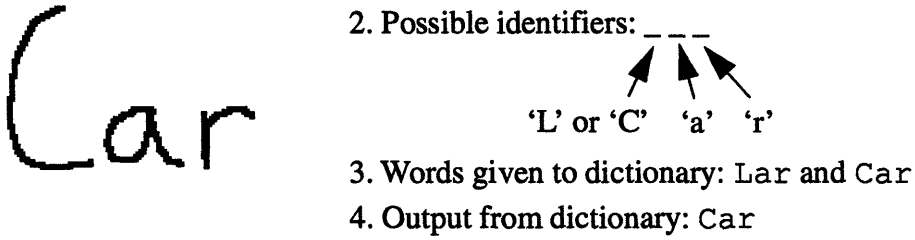


Figure 7.1 The word 'Car' is correctly identified using new method

The idea of using a threshold can also be applied to further improve these two recognition algorithms. Currently, these two algorithms recognize all written characters by selecting, for each input character, the identifier from the symbol database having the lowest similarity score. The identifiers are chosen regardless of how high the lowest similarity score is. It may sometimes occur, however, that no potential identifier from the symbol database is the proper identifier and thus no identifier receives a low similarity score. In this case, it is desirable to not recognize the poorly or incorrectly written input character at all. A better method for these algorithms is to have a threshold for which the lowest

similarity score must fall below in order to use that identifier as the recognized character. If no character from the symbol database is below the threshold, then the input character will not be recognized by the system and will be left as an empty space. This concept of using the threshold for similarity scores is even more important in off-line recognition algorithms where the user cannot interact with the system and correct the mistake, but even in on-line systems, it is sometimes more preferable to have the system not recognize ambiguous characters than to have it inaccurately recognize characters.

Future work can be done to perform even more sophisticated segmentation. This could perhaps allow even more flexible input such as the recognition of run-on discrete characters or even cursive script. Cursive script recognition is a very active field, but there has been no clear winner in the achievement of a highly accurate system. The cursive script recognition relies heavily on the preprocessing to standardize and normalize the characters being written. The cursive script recognition approaches have been to either break the character up into strokes and perform an approach similar to analysis by synthesis or to perform recognition of complete words. The implementation of a cursive script recognition system on the DSP would allow an even more flexible input and make the DSP even more attractive, but until an accurate system can be designed, no practical applications can be built and run with the DSP chip.

Finally, immediate future steps that can be made to these two real-time implementations in this thesis are to optimize the unboxed recognition system and improve the accuracy of segmentation. The unboxed recognition system is optimized by the C compiler using the optimization option, but no other optimizations are performed. In order to further increase the speed of the recognition, the assembly functions generated by the optimized C compiler can be rewritten, similar to the optimization process in the boxed recognition implementation. These rewritten assembly programs may result in a significant reduction in cycle times of the preprocessing and recognition functions. Because many of the preprocessing and recognition functions in the unboxed algorithm are the same as those used in the boxed recognition algorithm, the same optimized rewritten assembly programs for the boxed recognition may be used in the unboxed case, with possibly no or only a few minor revisions necessary. This would make the optimization process for the unboxed implementation easier, and the optimized preprocessing and recognition functions should result in fewer cycles spent in these stages. Furthermore, with the expected reduction in cycles when optimizing the context analysis and space recognition functions, the total recognition time per character in the unboxed case should drop dramatically.

7.3 Final Comments

The purpose of implementing the character recognition algorithms on the DSP chip is to use the DSP chip as the primary processor in the recognition process. This demonstrated capability and obvious improvement in the recognition speed without affecting the recognition accuracy makes the DSP chip an attractive processor for performing character recognition. The DSP is a very versatile chip that has already been proven to be very effective and efficient in telecommunications applications such as for modems, faxes, and

cellular phones. With the capability now of character recognition, the DSP chip can now serve as the processor for an integrated environment for very fast real-time applications such as that required by hand held computers and personal digital assistants. The DSP has the ability to handle e-mail, faxes, and model applications, and it has the ability to perform character recognition very accurately and quickly. These combined strengths makes the DSP an ideal platform for telecommunications applications and makes the DSP a useful and critical tool to be used in future computers and computer applications.

Appendix

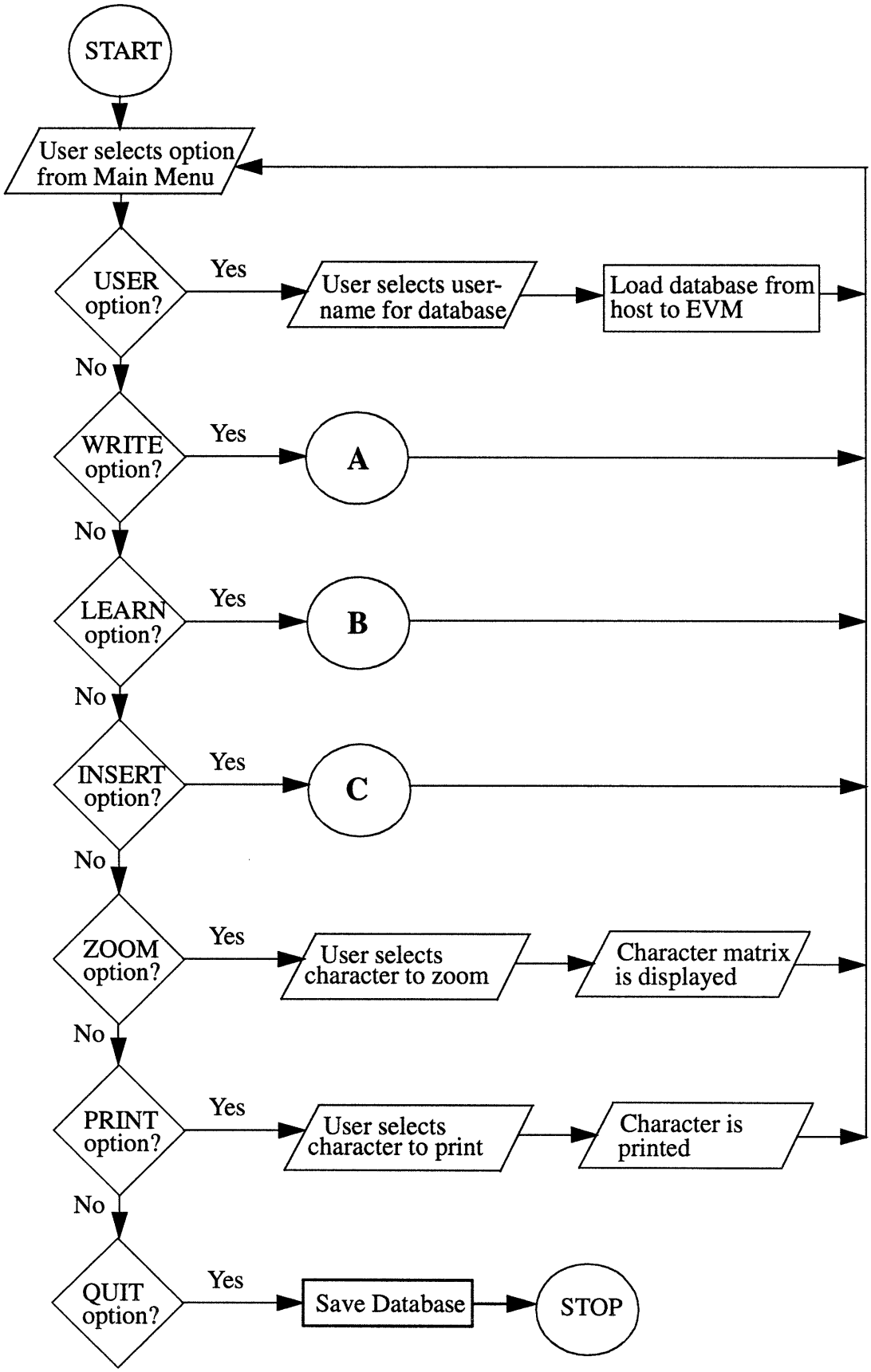
A.1 Recognized Characters Table

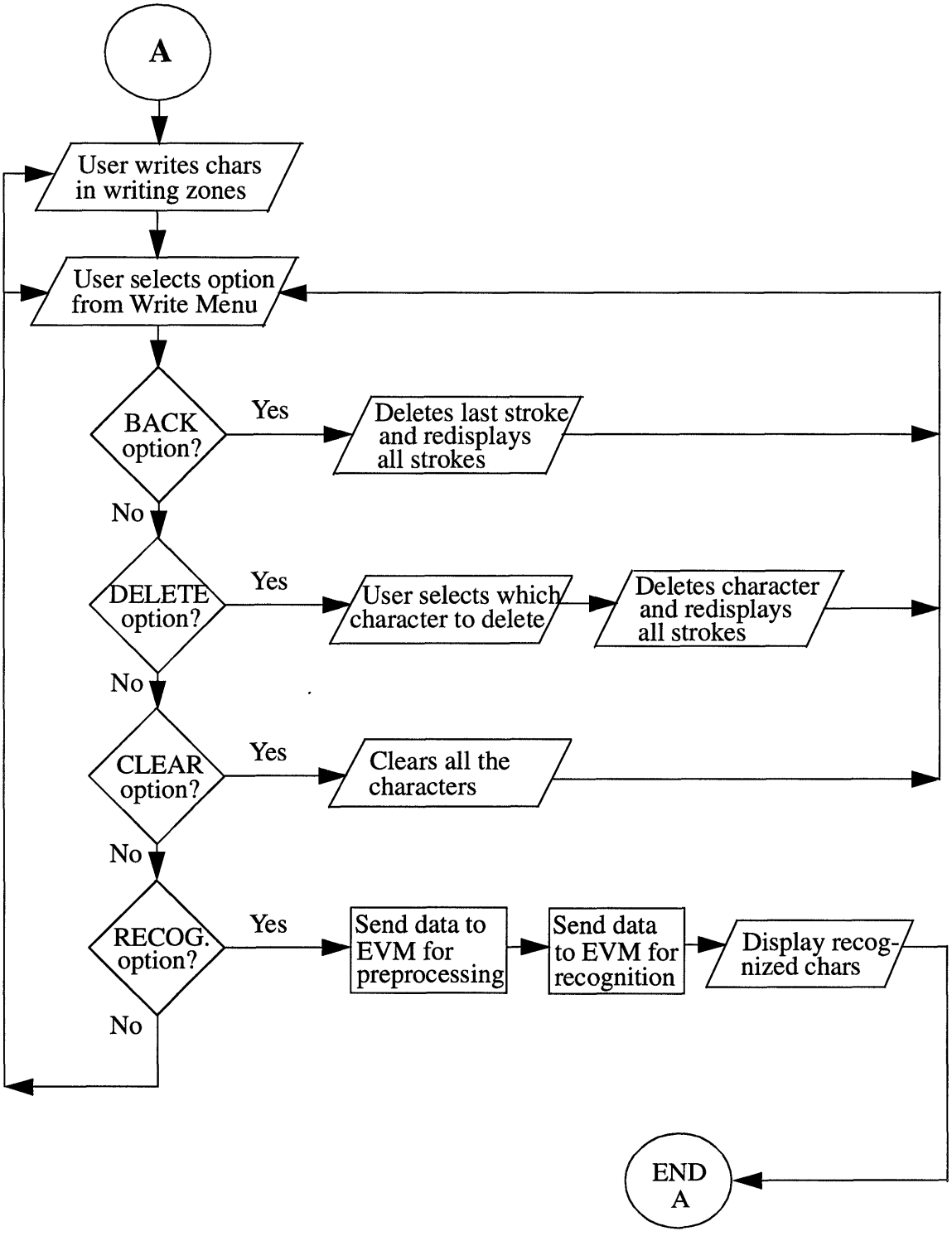
These are the recognized symbols that are learned and identified by both the boxed recognition algorithm and the unboxed recognition algorithm.

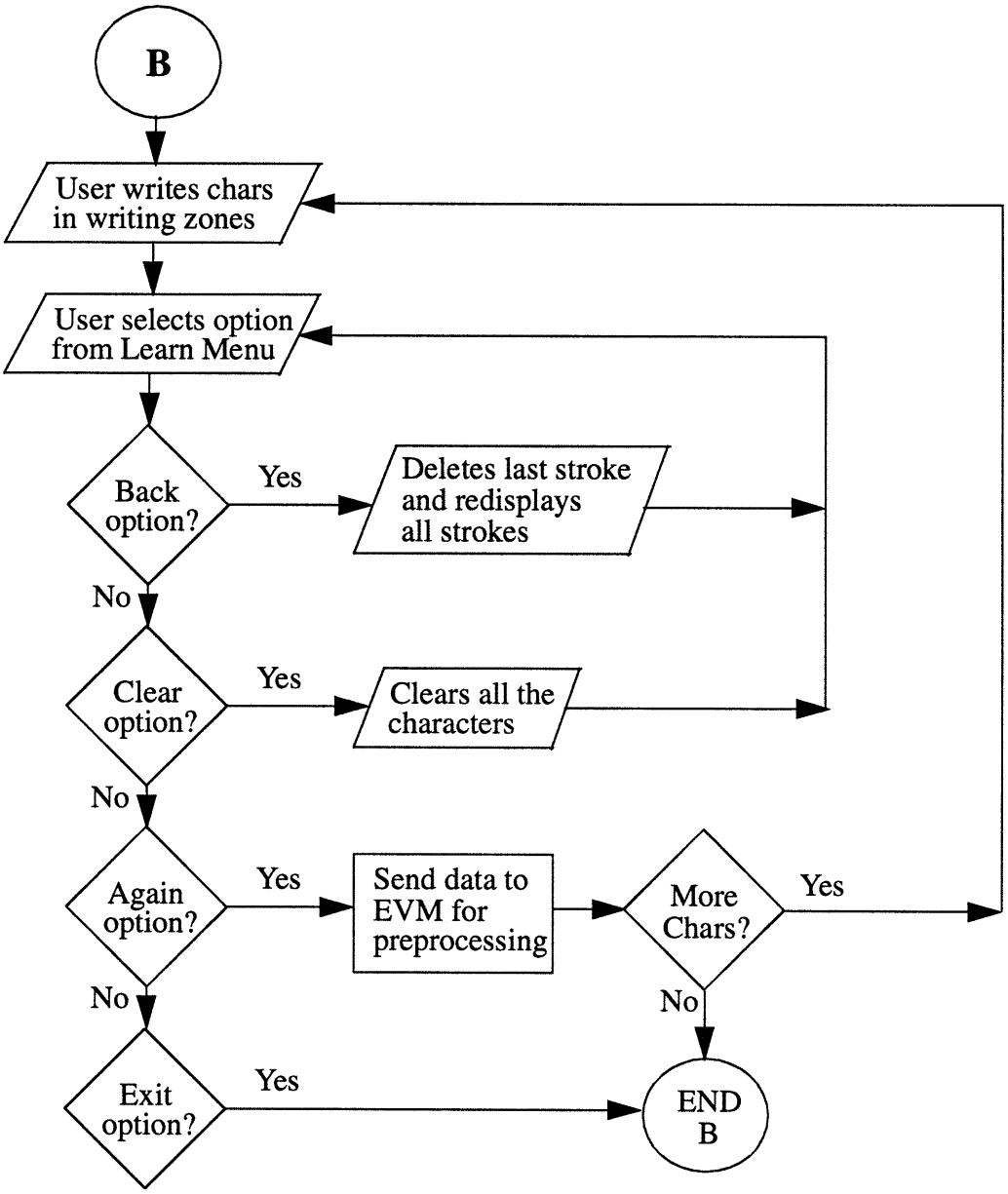
A	B	C	D	E	F	G	H	I
J	K	L	M	N	O	P	Q	R
S	T	U	V	W	X	Y	Z	a
b	c	d	e	f	g	h	i	j
k	l	m	n	o	p	q	r	s
t	u	v	w	x	y	z	0	1
2	3	4	5	6	7	8	9	‘
’	^	“	+	-	*	/	=	<
>	!	?	$	%	[	]	#	@

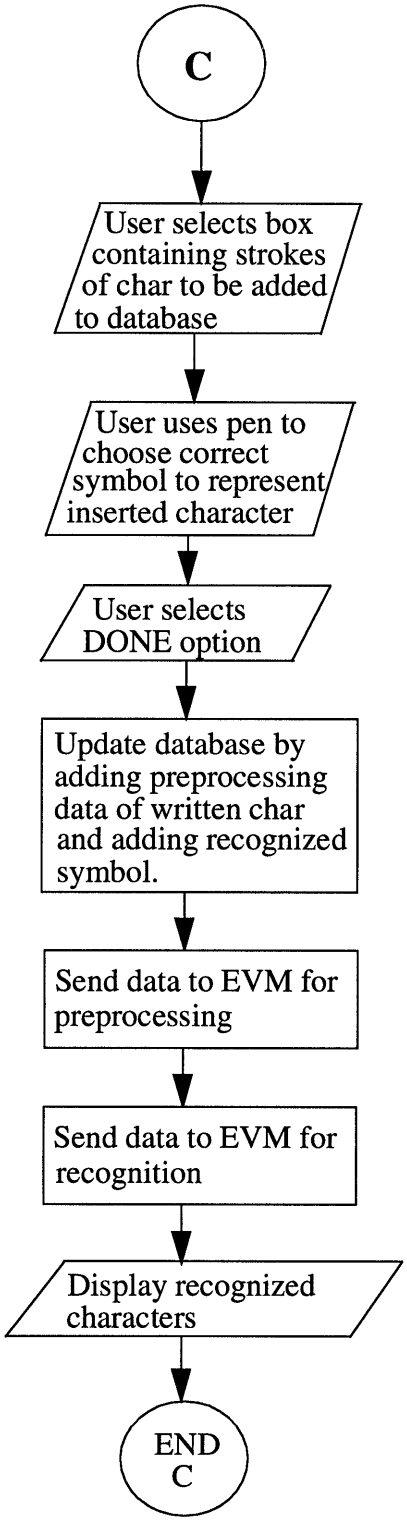
A.2 Boxed Recognition Algorithm Program Flowcharts

These flowcharts are designed to illustrate the typical program flow in the boxed recognition algorithm. Because the unboxed algorithm is so similar to the boxed algorithm and uses the same functions with the addition of three stages: segmentation, space recognition, and context analysis, the flowcharts for the unboxed algorithm are not shown. The same flowcharts as the boxed recognition algorithm can be used for the unboxed case with the simple addition of those three stages mentioned above.


START
User selects option from Main Menu
USER option?
Yes
User selects user-name for database
Load database from host to EVM
No
WRITE option?
Yes
A
No
LEARN option?
Yes
B
No
INSERT option?
Yes
C
No
ZOOM option?
Yes
User selects character to zoom
Character matrix is displayed
No
PRINT option?
Yes
User selects character to print
Character is printed
No
QUIT option?
Yes
Save Database
STOP


A
User writes chars in writing zones
User selects option from Write Menu
BACK option?
Yes
Deletes last stroke and redisplays all strokes
No
DELETE option?
Yes
User selects which character to delete
Deletes character and redisplays all strokes
No
CLEAR option?
Yes
Clears all the characters
No
RECOG. option?
Yes
Send data to EVM for preprocessing
Send data to EVM for recognition
Display recog-nized chars
No
END A


B
User writes chars in writing zones
User selects option from Learn Menu
Back option?
Yes
Deletes last stroke and redisplays all strokes
No
Clear option?
Yes
Clears all the characters
No
Again option?
Yes
Send data to EVM for preprocessing
More Chars?
Yes
No
Exit option?
Yes
END B
No


C
User selects box containing strokes of char to be added to database
User uses pen to choose correct symbol to represent inserted character
User selects DONE option
Update database by adding preprocessing data of written char and adding recognized symbol.
Send data to EVM for preprocessing
Send data to EVM for recognition
Display recognized characters
END C

References

[1] S. Mori, C.Y. Suen, and K. Yamamoto, "Historical Review of OCR Research and Development," *Proc. of the IEEE*, vol. 80, no. 7, pp. 1029-1058 (1992).

[2] G. Nagy, "At the Frontiers of OCR," *Proc. of the IEEE*, vol. 80, no. 7, pp. 1093-1100 (1992).

[3] T. Wakahara, H. Murase, and K. Odaka, "On-Line Handwriting Recognition," *Proc. of the IEEE*, vol. 80, no. 7, pp. 1181-1194 (1992).

[4] C. Barr and M. Neubarth, "Pen Pals," *PC Magazine*, vol. 12, no. 17, pp. 117-176, October 12, 1993.

[5] T. Halfhill, "PDAs Arrive But Aren't Quite Here Yet," *Byte*, vol. 18, no. 10, pp. 66-86, October 1993.

[6] C. Barr, "Pen PCs," *PC Magazine*, vol. 11, no. 19, pp. 175-203, November 10, 1992.

[7] C. Tappert, C.Y. Suen, and T. Wakahara, "The State of the Art in On-Line Handwriting Recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 8, pp. 787-808 (1990).

[8] S. Diehl and H. Eglowstein, "Tame the Paper Tiger," *Byte*, vol. 16, no. 4, pp. 220-230, April 1991.

[9] R. Davis and J. Lyall, "Recognition of Handwritten Characters," *Image and Vision Computing*, vol. 4, no. 4, pp. 208-218 (1986).

[10] C.Y. Suen, C. Nadal, R. Legault, T. Mai, and L. Lam, "Computer Recognition of Unconstrained Handwritten Numerals," *Proc. of the IEEE*, vol. 80, no. 7, pp. 1162-1180 (1992).

[11] H.L. Tan, "Hybrid Feature-Based and Template Matching Optical Character Recognition System," U.S. Patent 5,077,805, Dec. 31, 1991.

[12] "Recognition Technologies Under Discussion," *Electronic Documents*, vol. 2, no. 9, pp. 12-16 (1993).

[13] T. Pavlidis, "Recognition of Printed Text Under Realistic Conditions," *Pattern Recognition Letters*, vol. 14, no. 4, pp. 317-326 (1993).

[14] M.V. Nagendraprasad, P. Wang, and A. Gupta, "Algorithms for Thinning and Rethickening Binary Digital Patterns," *Digital Signal Processing*, vol. 3, no. 2, pp. 97-102 (1993).

[15] J. Pastor, "Optical Character Recognition by Detecting Geo Features," U.S. Patent 5,164,96, Nov. 17, 1992.

[16] D. Johnson, "Noise Tolerant Optical Character Recognition System," U.S. Patent 5,212,739, May 18, 1993.

[17] A. Nair and C.G. Leedham, "Preprocessing of Line Codes for Online Recognition Purposes," *Electronic Letters*, vol. 27, no. 1, pp. 1-2 (1991).

[18] W. Guerfali and R. Plamondon, "Normalizing and Restoring On-Line Handwriting," *Pattern Recognition*, vol. 26, no. 3, pp. 419-431 (1993).

[19] G. Boccignone, A. Chianese, L.P. Cordella, and A. Marcelli, "Recovering Dynamic Information from Static Handwriting," *Pattern Recognition*, vol. 26, no. 3, pp. 409-418 (1993).

[20] J. Mantas, "An Overview of Character Recognition Methodologies," *Pattern Recognition*, vol. 19, no. 6, pp. 425-430 (1986).

[21] P. Gader, B. Forester, M. Ganzberger, A. Gillies, B. Mitchell, M. Whalen, and T. Yocum, "Recognition of Handwritten Digits Using Template and Model Matching," *Pattern Recognition*, vol. 24, no. 5, pp. 421-431 (1991).

[22] Y. Lu, S. Schlosser, and M. Janeczko, "Fourier Descriptors and Handwritten Digit Recognition," *Machine Vision and Applications*, vol. 6, no. 1, pp. 25-34 (1993).

[23] M. Lai and C.Y Suen, "Automatic Recognition of Characters by Fourier Descriptors and Boundary Line Encodings," *Pattern Recognition*, vol. 14, no. 1, pp. 383-393 (1981).

[24] M. Shridhar and A. Badreldin, "High Accuracy Character Recognition Algorithm Using Fourier and Topological Descriptors," *Pattern Recognition*, vol. 17, no. 5, pp. 515-524 (1984).

[25] P. Trahanias, K. Stathatos, F. Stamatelopoulos, and E. Skordalakis, "Morphological Hand-Printed Character Recognition by a Skeleton Matching Algorithm," *Journal of Electronic Imaging*, vol. 2, no. 2, pp. 114-125 (1993).

[26] A. Rajavelu, M.T. Musavi, and M.V. Shirvaikar, "A Neural Network Approach to Character Recognition," *Neural Networks*, vol. 2, no. 5, pp. 387-393 (1989).

[27] G. Martin and J. Pittman, "Recognizing Hand-Printed Letters and Digits Using Backpropagation Learning," *Neural Computation*, vol. 3, no. 2, pp. 258-267 (1991).

[28] D. Kelly, "Neural Networks for Handwriting Recognition," *Proceedings of the SPIE-The International Society for Optical Engineering*, vol. 1709, pt. 1, pp. 143-154 (1992).

[29] A. Kundu, Y. He, and P. Bahl, "Recognition of Handwritten Word: First and Second Order Hidden Markov Model Based Approach," *Pattern Recognition*, vol. 22, no. 3, pp. 283-297 (1989).

[30] Y. He, M.Y. Chen, A. Kundu, "Handwritten Word Recognition Using HMM with Adaptive Length Viterbi Algorithm," *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 3, pp. 153-156 (1992).

[31] *TMS320C5x User's Guide*, Texas Instruments, 1993.

[32] *TMS320C5x Evaluation Module Technical Reference*, Texas Instruments, 1992.

[33] *TMS320C2x/C5x Optimizing C Compiler User's Guide*, Texas Instruments, 1991.